X68k

Programming Series

吉野智興＋中村祐一＋石丸敏弘＋今野幸義＋村上敬一郎＋大西恵司

(♯0)

X680x0 Develop. & libc II

X 6 8 k

Programming Series

吉野智興＋中村祐一＋石丸敏弘＋今野幸義＋村上敬一郎＋大西恵司

(#0)

X680x0 Develop. & libc II

本書に付属するディスクに収録されている開発環境は，**X68000** および **X68030** で動作します。この開発環境が快適に動作するためには，最低 2M バイトのメモリと 6M バイトのハードディスクが必要です。

本書は「***X68k Programming Series #1 X68000 Develop.***」および「***X68k Programming Series #2 X680x0 libc***」の各機能を強化するとともに **X68030** にも対応させたものです。
基本的な機能・仕様については「***X68k Programming Series #1 X68000 Develop.***」「同 ***Manual Books***」および「***X68k Programming Series #2 X680x0 libc***」「同 ***Manual Books***」を参照してください。

1-1 インストールの準備

実際にインストールを始める前に，GCC/LIBC に必要な環境設定やディスク容量の確認をしておきましょう。説明をよく読んでインストールの準備をしてください。

1-1-1 ディスクスペースの準備

GCC/LIBC は，全部で 3 つのパッケージで構成されています。まず 1 つはコンパイラパッケージです。ここには，コンパイラやアセンブラ，リンカ，デバッガの実行環境が含まれます。2 つ目はライブラリパッケージです。ライブラリに，XC の代わりに LIBC を利用するためのインクルードファイルやライブラリファイルが含まれます。最後の 3 つ目はソースコードパッケージです。LIBC のすべてのソースコード，付属資料などライブラリパッケージ以外のものが含まれます。

これら 3 つのパッケージは，それぞれインストールするかどうか選択できるようになっています。それぞれのパッケージは概算で，Table 1 に示されるだけのディスク容量を必要とします。あなたが必要とするパッケージとディスクの空き容量を見て，どれをインストールするかを決めてください。

Table 1● インストールに必要なディスク容量

パッケージ名	インストール容量
コンパイラパッケージ	1.2M バイト
ライブラリパッケージ	1.0M バイト
ソースコードパッケージ	3.5M バイト

たとえば，コンパイラパッケージとライブラリパッケージの 2 つをインストールするには，最低限 2M バイトのメモリと 2.5M バイトのディスク容量が必要であることがわかります。これらのパッケージをそれぞれ別々にインストールすればフロッピーディスクで運用することも不可能ではありませんが，できればハードディスクの方が望ましいでしょう。ただし，ソースコードパッケージのインストールには，そのサイズからハードディスクが必ず必

要となります。

もし残りディスク容量がここで示した容量に満たない場合は，インストールプログラムが警告を表示して処理を中断しますから，別のディスクを指定するか，パッケージの選択をやりなおしてください。

1-1-2 環境の準備

GCC/LIBC をインストールし，実行させるにはいくつかの環境設定を行う必要があります。これらの新たな設定については，インストールプログラムがあなたの環境に合わせて参考となる設定ファイル "AUTOEXEC.NEW" を作成しますから，それをもとに "AUTOEXEC.BAT" を自分で編集しなおす必要があります[1]。

また，ソースコードパッケージをインストールする場合は，これに加えて特別な環境設定が必要になります。ソースコードパッケージ内のファイルのなかには非常に長いファイル名のものが含まれており，そのままインストールするといくつかのファイルが重複して消えてしまいます。これを防ぐためにソースコードパッケージをインストールする前に，ファイル名を 21 文字すべて認識させるようなプログラム "TwentyOne.X" を常駐させなければなりません[2]。

1) 詳しくは「AUTOEXEC.BAT の書き換え」(P.ix) でもう一度解説します。

2) 詳しくは「特別な環境設定」(P.vi) でもう一度解説します。

1-2 インストールする

準備はもう整ったでしょうか。それではいよいよインストール作業に入ります。説明をよく読んで，まちがいのないように気をつけてください。

1-2-1 起動する

まず，X68000 または X68030 の電源を入れ，あなたが普段使っている環境で立ち上げてください。フロッピーディスクだけで使用している人は，そのシステムディスクから，またハードディスクを使っている人はそのハードディスクから Human68k を起動してください。また，あなたが SX-Window を使っているのならば，SX-Window を終了させて，コマンドラインに戻るか，“COMMAND.X” のアイコンをダブルクリックしてコマンドシェルを起動してください。

付属のディスクからは起動できません

次に，GCC/LIBC に付属しているディスクを取り出します[1]。そしてそのディスクをフロッピーディスクドライブに挿入し，カレントドライブをそのディスクドライブに変更してください。たとえば，“C:” ドライブにディスクを挿入した場合は次のようにタイプします。画面の表示例中の “A:¥>” という部分は Human68k のプロンプトです。

1) 安全のため，ディスクは必ずバックアップをとり，バックアップしたディスクをご使用ください。

```
A:¥>C:
```

最後に，そこから “SETUP.X” を起動します。次のようにタイプしてください。

```
C:¥>SETUP
```

1-2-2 インストールを始める

正しく“SETUP.X”が実行されると，画面にインストールプログラムのメッセージが表示されます。基本的なインストールの方法は非常に簡単なものです。画面の質問の意味をよく理解して，正しく答えるようにしてください。

なお，画面の表示例中の CTRL+C とは，CTRL キーを押しながら C を押すことを意味します。

```
****************************************************************
*                GCC/LIBC - 付属ディスク SETUP                 *
****************************************************************

※今から GCC/LIBC の各パッケージを付属ディスクからあなたのハードディス
クあるいはフロッピーディスクにインストールします。これから画面に出され
る質問に正しく答えてください。

問>GCC/LIBC のパッケージのうち，どのパッケージをインストールしますか？
次の 3 つのうちから選択し，その番号を入力してください。複数ある場合はス
ペースで区切って指定してください。

なお，中断するには CTRL+C を押してください。

     <1> コンパイラパッケージ     (ディスク容量 1.2MB 必要)
     <2> ライブラリパッケージ     (ディスク容量 1.0MB 必要)
     <3> ソースコードパッケージ   (ディスク容量 3.5MB 必要)

答>
```

まずインストールするパッケージを選択してください。この選択をする前にハードディスクのどのディレクトリにインストールするか，またそのディレクトリの空き容量が十分にあるかどうかについて調べておいてください。

なお，ソースコードパッケージをインストールする場合には，次項に示す「特別な環境設定」が必要となりますので注意してください。

1-2-3 特別な環境設定

あなたがもしソースコードパッケージをインストールするように指定した場合は，この「特別な環境設定」が必要となります。ソースコードパッケージをインストールしない場合はここを読む必要はありません。次へ進んでください。

さて「環境の準備」(P.iv) の部分でも触れましたが，ソースコードパッケージを正しくインストールするためには，ファイル名を 21 文字まで正しく認識させるように，あらかじめ環境設定しなくてはなりません。

Human68k はファイル名を 21 文字まで扱えるにもかかわらず，標準のままではそのうちの先頭の 8 文字までしか実際には認識しません。そのため，9 文字目以降が異なっていても先頭さえ同じなら，別のファイルでも同じファイルだと認識してしまいます。ソースコードパッケージ中のファイルは非常に長いファイル名が多いので，そのままではいくつかのファイルの名前が上記の理由で重複してしまいます。

この問題を解決するためには，“TwentyOne.X” というフリーウェアを利用します。このソフトは付属のディスクに収録されています。もしソースコードパッケージをインストールするように選択した場合は，インストールプログラムが自動的にあなたの環境をチェックし，“TwentyOne.X” を必要に応じて組み込むようになっています。もちろん，すでに組み込まれている場合はこの処理は行われません。次へ進んでください。

あなたの環境に “TwentyOne.X” がインストールされていない場合は，次の質問が表示されます。新たに “TwentyOne.X” にインストールするディレクトリを指定してください。

```
問>あなたの環境には "TwentyOne.X" がインストールされていません。ソースコードパッケージを展開するためには，これをインストールすることが必要です。この常駐プログラムをどこにインストールしますか？

インストールするディレクトリをフルパスで指定してください。

なお，中断するには CTRL+C を押してください。

答>
```

たとえば，“A:” ドライブの “¥bin” というディレクトリにインストールしたいときは，次のようにタイプしてください。

```
答>A:¥bin
```

インストールプログラムは，指定されたディレクトリに適切なバージョンの “TwentyOne.X” をコピーし，それに合わせて環境設定の参考例をファイルに書き出します。

```
※あなたの環境に合わせた環境設定の参考例を AUTOEXEC.NEW として作成しました。ソースコードパッケージを正しくインストールするにはこれを参考にして
AUTOEXEC.BATを編集しなおし，マシンを再起動させる必要があります。

インストールはここで一旦終了です。

再起動した後にもう一度 SETUP.X を実行してください。

C:¥>
```

1-2-4　インストール先の指定

次にインストールするディレクトリを指定してください。インストールプログラムは指定されたディレクトリにすべてのデータをインストールできるかどうか，空き容量をチェックし，十分な空き容量があればインストールを開始します。

```
問>指定したパッケージをどこにインストールしますか？
インストールするディレクトリをフルパスで指定してください。

なお，中断するには CTRL+C を押してください。

答>
```

たとえば，“**A:**”ドライブの“**¥lang**”というディレクトリにインストールしたいときは，次のようにタイプしてください。

```
答>A:¥lang
```

ここで，もしディスクの空き容量が足りない場合は次のようなメッセージが表示され，インストールプログラムは中断します。インストールするドライブ，ディレクトリをもう一度確かめてから，インストールを最初からやりなおしてください。

```
※指定したディレクトリには選択したパッケージをインストールするだけの十分な
空き容量がありません。

もう一度確認して最初からインストールをやりなおしてください。

C:¥>
```

1-2-5 インストール実行

ディスクの空き容量の確認が終わると，インストールプログラムはパッケージのインストールを開始します。所要時間は選択したパッケージによりますが，長い場合には約5，6分ほどかかりますので，お茶でも飲みながらしばらく作業を見守っていてください。以下の画面表示例では，コンパイラパッケージおよびライブラリパッケージのインストールを想定しています。

```
※今から選択したパッケージをインストールします。

 Mode    Size     Date      Time Name
-arw-   29636 94-07-28 14:57:10 bin/gcc.x
                    :
                    :
                  (中略)
                    :
                    :
-arw-    2872 94-07-28 23:49:12 lib/libtz.a
-arw-    4720 94-07-28 23:49:18 lib/libw.a
```

1-2-6 AUTOEXEC.BAT の書き換え

ここまでで，指定したパッケージすべてが展開されています。インストールプログラムは，これらのパッケージを使用するための環境設定をあなたの環境に合わせて自動的に作成し，それを“AUTOEXEC.BAT”と同じディレクトリにファイルとして書き出します。

インストールプログラムは実際に“AUTOEXEC.BAT”を書き換えることまではしませんから，あとはあなたがこの作成された環境設定ファイル“AUTOEXEC.NEW”を参考にして，自分で“AUTOEXEC.BAT”を編集してください。このファイルの中身のうちの必要な部分だけをカットアンドペーストすれはよいよう

になっています。

```
※あなたの環境に合わせた環境設定の参考例を "AUTOEXEC.NEW" として作成しま
した。各自でこれを参考にして "AUTOEXEC.BAT" を編集しなおしてください。

これでインストールはすべて終了です。
おつかれさまでした。

C:¥>
```

Develop. & libc II

Chapter 0

Chapter 1

Chapter 2

Chapter 6

Chapter 7

Chapter 8

Chapter 9

本書で引用，表記している「*X68000 Develop.*」は，前著「*X68000 Develop.*」および新著「*X680x0 Develop. Manual Books*」を指しています。同様に「*X680x0 libc*」は，前著「*X680x0 libc*」および新著「*X680x0 libc Manual Books*」を指しています。

本書で参照する場合，それぞれの前著と新著の該当ページは基本的に同じですが，改訂内容によっては異なることもあります。その場合は，そのつど明記してあります。

X68k

Programming Series

(#1)

X680x0 Develop.

はじめに

前著「*X68k Programming Series #1 X68000 Develop.*」を準備をしている時期が，**SHARP** から新機種 **X68030** がでるという噂が流れていたころでした。**X68030** は，本当に CPU を 32 ビット化しただけの内容にとどまりましたが，それでも初代の数倍の処理速度になっています。またリスクは伴いますが，**X68030** に 68040 を登載するボードも登場し，これを装着した **X68030** では，初代の十倍程度の処理能力を出すこともあります。

パワーアップしたハードウェアの性能を十分に引き出せるように，コンパイラ，アセンブラ，リンカ，デバッガを，この新しいハードウェアに合わせてバージョンアップしたものが本書「*X680x0 Develop. & libc II*」です。とはいえ，前著の「*X68000 Develop.*」に比べると実使用時間が短いために，まだ完全ではない可能性も否定できません。しかし，できるだけ多くの **X680x0** ユーザにとって，より理想的な開発が可能なように仕上げたつもりです。

X680x0 は，本当にユーザが「楽しんでプログラムできる」国産ただ一つのパソコンだと信じています。その楽しいプログラミングに本書と付属するツール類が役にたてば，非常にうれしいと思います。

1994 年 8 月

吉野　智典

Chapter 1

X680x0 GCC

本章では，X680x0 GCC の変更点について説明します。すでに X68000 GCC の機能についてある程度知識があることを前提にして，変更点についてのみ触れます。この他の機能については，「*X68k Programming Series #1 X68000 Develop.*」または「*X680x0 Develop. Manual Books*」を参照してください。

1-1 X680x0 GCCの拡張概要

従来の **X68000 GCC** との最大の相違点として，**SHARP** 純正のアセンブラではアセンブルできないコードを生成するようになったことがあげられます。

従来は純正のアセンブラも意識したコードを生成するようにしていましたが，現時点の最新版の純正アセンブラは，68020 コードで **X680x0 GCC** のソースを正しくアセンブルできないバグがあることやアセンブル速度の面でサポートする価値が薄いと判断したため，サポートを断念しました。実は，すでに **SX-Window** 対応モードを装備した時点で，純正アセンブラではアセンブルできない状態だったのですが……。今回の企画によって，本書付属の **X680x0 HAS** や **X680x0 HLK** も純正製品の仕様からかなり離れてしまいましたから，**X680x0 GCC** のサポート範囲から純正アセンブラを除外しても許されると判断しています。この点以外は，**X680x0 GCC** は従来の **X68000 GCC** の機能をほぼそのまま踏襲し，以下のような改良を加えています。

◆対象 CPU の拡張

・**X68030** での 68030 コードだけでなく，64040 のコード[1]も生成可能です。

1) 68030 + 68881/68882 モードで生成されたコードの一部は，68040 では実行できません。

◆予約語の追加

・従来の **X68000 GCC** では特殊内部関数の呼び出しで行っていた割り込み処理関数を，**MS-DOS** コンパイラで実装されている `interrupt` 予約語を使用する形式に変更しました[2]。

・変数のアクセスに，すべてプログラムカウンタ間接アドレッシングを強制する予約語を新設しました。

2) 従来の方法も使用できます。

これらの変更は，コンパイル時に指定するオプションスイッチと予約語の変更/追加から実現されています。また，今までに報告されたコンパイラの最適化ミスなどのバグも修正してあるので，より安定したコードが生成できるようになりました。

1-2 予約語の変更

X680x0 GCCには，たくさんの予約語が追加されています。このなかで本当に意味があるのは，「割り込み関数宣言の予約語」と「プログラムカウンタ間接強制の予約語」にあげられている予約語だけです。最後の「簡便のために追加された予約語」は，オプションスイッチの指定でまったく無視される予約語です。

1-2-1 割り込み関数宣言の予約語

従来のX68000 GCCで使用されていた __builtin_saveregs ()[1]を用いた割り込みをハンドルする関数を，MS-DOSのC言語のように，宣言ででも使えるようになっています。そのための予約語が次の3つです。

1) 「*X68000 Develop.*」Vol.I P.50を参照してください。

- __interrupt__
- __interrupt
- interrupt

文法上はstaticやexternと同じ実装になっています。しかし，従来のX68000 GCCで __builtin_saveregs () で示していた単純割り込み処理関数が，この宣言で有効になります。

List 1-1 __interrupt__予約語のサンプル

```
1:  extern void foo(void);
2:
3:  void __interrupt__
4:  intfunc(void)
5:    {
6:      foo ();
7:    }
```

List 1 2 List 1 1のコンパイル結果例

```
1:          .globl _intfunc
2:  _intfunc:
3:          movem.l d0/d1/d2/a0/a1/a2,-(sp)
4:          jbsr _foo
5:          movem.l (sp)+,d0/d1/d2/a0/a1/a2
6:   rte
```

1-2-2 プログラムカウンタ間接強制の予約語

次の3つの予約語が指定された変数は，プログラムカウンタ間接アドレッシングでアクセスすることをコンパイラに強制します。この強制は変数のみに限定されています。すべてのメモリ番地の参照にプログラムカウンタ間接を強制させるには，同時に -fall-bsr スイッチ[2]を指定してください。

2) 「PC間接形式の使用」(P.11) を参照してください。

- __pcr__
- __pcr
- pcr

生成コードの質は若干劣悪になりますので，スピードを重視する場合にはポインタを多用するか，潔くアセンブラで書き替えを行うのが速度を稼ぐよい方法です。

List 1 3 __pcr__予約語のサンプル

```
1:  __pcr__ struct {
2:    int x;
3:    int y;
4:    int val;
5:  } pcr_data;
6:
7:  void
8:  ini_data ()
9:  {
10:   pcr_data.x = 0;
11:   pcr_data.y = 0;
12:   pcr_data.val = 0;
13: }
```

List 1 4 List 1 3のコンパイル結果例

```
 1:         .globl _ini_data
 2:  _ini_data:
 3:         lea _pcr_data.w(pc),a0
 4:         moveq.l #0,d0
 5:         move.l d0,(a0)
 6:         lea _pcr_data.w(pc),a0
 7:         move.l d0,4(a0)
 8:         lea _pcr_data.w(pc),a0
 9:         move.l d0,8(a0)
10:         rts
11:        .xdef _pcr_data
12:  _pcr_data:
13:        .ds.b 12
```

pcr 宣言された変数は，テキストセクション[3)]に出力されることに注意してください。テキストセクションではシンボルの多重定義ができませんから，リンケージは従来の **UNIX** 互換から **XC** 互換になります。そのため適切な extern 宣言を欠いていると，リンクする際，エラーになることがあります。

3) 「*X68000 Develop.*」Vol.I P.87 を参照してください。

1-2-3 簡便のために追加された予約語

MS-DOS 上のコンパイラで作成されたソースを簡便にコンパイルするには，-fms-dos スイッチが利用できます。その際，以下の予約語が無視されます。

- far
- __far
- near
- __near
- huge
- __huge
- cdecl
- __cdecl
- pascal
- __pascal

1-3 その他の拡張

前節で解説した以外に，X680x0 GCC で追加された事項について補足しておきます。

1-3-1 Human68k のバージョン

X680x0 GCC は稼動環境の Human68k のバージョンを取得して，アセンブラのラベル RUNS_HUMAN_VERSION にそのメジャーバージョン番号をセットします。稼動環境の Human68k のバージョンが 3.02 であれば 3, 2.01 であれば 2 が RUNS_HUMAN_VERSION に設定されます。

1-3-2 .cpu 疑似命令[1]

1) 「アセンブル対象命令セットの指定」(P.41) を参照してください。

HAS に与える .cpu 疑似命令は -m68020 スイッチなどで与えられる CPU タイプに依存しています。

asm 文で命令を直接記述する場合に，CPU のタイプと記述するコードが一致していないと，HAS がエラーになる場合があります。

Table 1-1● .cpu 擬似命令と CPU タイプ

CPU タイプ	.cpu 擬似命令
default	.cpu 68000
-m68020	.cpu 68030
-m68040	.cpu 68040

1-3-3 境界整合について

X68030 は本当の 32 ビットバスになっているため，プログラムやデータは 4 の倍数で整合しているのが最も速いアクセスになります。X680x0 GCC では，デフォルトでは -m68020 スイッチなどの上位 CPU のオプションスイッチが与えられた場合でも .even 疑似命令[2]を生成します。

2) 「*X68000 Develop.*」Vol.1 P.116 を参照してください。

より **X68030** で高度な速度を要求する場合には，環境変数に境界整合のための疑似命令を表す文字列を設定しておけば，任意の境界にプログラムやデータを整合させることができます。

◆環境変数 GCC_TEXT_ALIGN

関数の冒頭に，出力される境界整合疑似命令文字列を設定しておきます[3]。

3)「アドレス境界の調整」(P.46) を参照してください。

```
set GCC_TEXT_ALIGN=.quad
```

◆環境変数 GCC_BLOCK_ALIGN

無条件ジャンプ命令の直後のラベルなどのプログラム上でのブロックの冒頭に，出力される境界整合疑似命令を設定します。

◆環境変数 GCC_DATA_ALIGN

データセクションでの境界整合に使われる疑似命令文字列を設定しておきます。

これらの環境変数は，-m68020 または -m68040 スイッチが指定された場合にだけ参照され，設定されていればその文字列を境界整合のための疑似命令として出力します。当然ですが，生成されるコードは従来より大きなものになります。しかし，**X680x0 GCC** のようにポインタを多用した巨大なアプリケーションの場合，-m68020 スイッチ指定によるコード縮小が，この境界整合のためのメモリのムダを上回ってコードが小さくなる場合もあります。

1-4 拡張されたオプションスイッチ

X680x0 GCC では，X68000 GCC からより機能を高めるために，いくつかのオプションスイッチを変更/追加/廃止しています。

1-4-1 廃止されたオプションスイッチ

●`-fall-jsr`　　すべての関数をロングコールで指定

●`-fstrings-nopcr`　文字列のプログラム相対禁止

この 2 つのオプションスイッチは，HAS の改良によってプログラムカウンタ間接アドレッシング[1]がより柔軟になったために廃止されました。

1) Appendix A「アドレス形式の表記」(P.160) を参照してください。

同一ソースプログラム内に存在する文字列のアドレスや関数の相互呼び出しは，可能であればすべてプログラムカウンタ間接に，16 ビットオフセットで届かない場合は絶対アドレスに，HAS が自動的に振り分けを行います。この機能を X680x0 GCC は使用していますので，従来の X68000 GCC でアセンブルエラーになっていた巨大なプログラムでも問題なくアセンブルできます。さらに，従来の X680x0 では不可能だった同一ソースファイル内の関数の前方参照の最適化が行われるため，より小さなオブジェクトを生成するようになりました。

●`-as-symbols`　　アセンブラの最大シンボル数の指定

HAS の改良によって，シンボル数の制限がなくなりました。そのため，実質上指定する意味がなくなったオプションスイッチです。仮に指定しても，何の動作もしません。

1-4-2 変更されたオプションスイッチ

新しく追加された予約語 pcr [2]を生かすために，若干意味が変わったオプションスイッチがあります。

2)「プログラムカウンタ間接強制の予約語」(P.6) を参照してください。

●`-fall-bsr`　PC 間接形式の使用

-fall-bsr　PC 間接形式の使用

書式：　-fall-bsr[3]

機能：　分岐命令をすべてプログラムカウンタ間接にします。

解説：　従来の -fall-bsr スイッチは，ただ単に関数を呼び出す場合に jsr に代えて bsr を指定するだけでした。しかし **X680x0 GCC** からは，プログラムコードのアドレスを扱う場合には，すべてプログラムカウンタ間接命令の使用を強制するようになりました。新しく設けられた予約語 pcr との併用で，16 ビットで届く範囲であれば，完全にリロケータブルなコードが生成可能になっています。

3)「*X68000 Develop.*」Vol.2 P.35 を参照してください。

1-4-3 追加されたオプションスイッチ

以下のようなオプションスイッチが追加されています。

●-m68020	68020/68030 用コードの生成
●-m68040	68040 用コードの生成
●-fansi-only	ANSI で規定された予約語のみ認識
●-ffppp	FPPP.X 用コードの生成
●-ffpu-hard-bug	ハード障害を回避するコードの生成
●-fignor-cpu-type	CPU チェックコードを挿入せずにコンパイルする
●-flong-offset	PC 間接命令をすべてロングオフセットにする
●-fms-dos	8086 系 **C** プログラムをコンパイルする
●-fpic	変数をすべて A5 ベースの間接アドレッシングにする

-m68020　68020/68030 用コードの生成

書式：　-m68020

機能：　生成コードを 68020/68030 用にします。

解説：　ターゲット CPU として 68020 を想定したコード生成を行います。オプションスイッチ名称は “-m68020” ですが，実行コードは 68030 と 68040 で実行可能です。このオプションスイッチが指定された場合には，以下のような 68020，68030，68040 でのみ実行可能なコード生成が行われます。

●整数演算では演算のためのライブラリコールが行われない

●オプションスイッチによって 32 ビットフルリロケータブルコード[4]が生成可能

●メモリ間接アドレッシングモードを使ったほうが速いときに，このアドレッシングモードを使う場合がある

このオプションスイッチを使用した場合には，-SX スイッチ[5]か -fignor-cpu-type スイッチ[6]が指定されていない条件で，関数 main () に CPU チェックコードが付加され，**X68000** では実行できないプログラムが生成されます。

CPU チェックは，ハード改造された **X68000** の場合も考慮して外部のワーク等を参照せず，スケールファクタをもったアドレッシングモードが 68000 では無視されることを利用して，CPU を判別しています。

4) ライブラリ (**XC** および **LIBC**) は，フルリロケータブルになっていない点に注意してください。

5) 「*X68000 Develop.*」Vol.2 P.43 を参照してください。

6) 今回新しく追加したオプションスイッチです。「CPU チェックコードを挿入せずにコンパイル」(P.14) を参照してください。

-m68040　68040 用コードの生成

書式： -m68040

機能： 生成コードを 68040 用にします。

解説： ターゲット CPU として 68040 を想定します。このオプションスイッチを指定した場合には -m68881 スイッチ[7]が同時に有効になりますが，68040 では実行できない FPU コードの生成を避けるようになります。このため生成されるコードが，-m68020 と同時に -m68881 スイッチを指定した場合と異なることがあります。

68040 では -m68020 と同時に -m68881 スイッチを指定して生成されたコードは実行できないので，FPU を装着した **X68030** をターゲットにしたプログラムを作成する場合には -m68020 と -m68881 スイッチでなく，-m68040 スイッチを使用するようにしてください。

7) 「*X68000 Develop.*」Vol.2 P.18 を参照してください。

-fansi-only　ANSI で規定された予約語のみ認識

書式： -fansi-only

機能： ANSI で規定された予約語のみ認識します。

解説： **X680x0 GCC** が認識する予約語で拡張されたものはたくさんあります。しかし，しばしばこの拡張された予約語が変数名と衝突を起こすので，そのような衝突を避けるために用意されたオプションスイッチです。

このオプションスイッチを指定すると，拡張された予約語は“_”で始まるものだけしか予約語として認識しなくなります。このオプションスイッチで無効になる予約語は以下のとおりです。

- remote
- relocate
- common
- SXCALL
- DOSCALL
- asm
- inline
- typeof
- interrupt
- pcr
- huge
- far
- near
- cdecl
- pascal

-ffppp　FPPP.X 用コードの生成

書式： -ffppp

機能： FPPP.X 用のコードを生成します。

解説： **HAS** が浮動小数点数を直接扱えるようになりましたので，**GCC** 側では浮動小数点数を従来のように 16 進数に直すことを行わなくなりました。

しかし，従来の **X68000** で FPPP.X を使用した場合には，FPPP.X がこの浮動小数点数を認識しないため，エラーになります。FPPP.X を使用する場合には，このオプションスイッチを指定してください。

-ffpu-hard-bug　ハード障害を回避するコードの生成

書式： -ffpu-hard-bug

機能： VRAM アクセスにともなう障害を回避するコードを生成します。

解説： **X68030** には CPU として **MC680EC30** が登載されています。その CPU を **MC68030** などに交換した場合，グラフィック VRAM に対して書き込みを行った直後にメモリをアクセスする FPU 命令を実行すると，正しく実行されないハード障害が起こることがあります。

グラフィック VRAM を直接アクセスし，同時に FPU を操作する場合にはこのオプションスイッチを指定してコンパイルしてください。コンパイラは，メモリライトアクセスの直後の FPU 命令がメモリを操作する場合には，nop コードを挿入してハード障害を回避します。

ただし，プログラム全体に対してこのオプションスイッチを指定するのは実行速度上抵抗がある場合には，#pragma[8] を使用して，必要な範囲についてのみ有効にすることも可能です。

8) 「*X68000 Develop.*」Vol.I P.57 を参照してください。

このハード障害は少なからず報告されていますので，このような種類のプログラムを一般に公開される場合は，このオプションスイッチを指定されることを希望します。

-fignor-cpu-type　CPU チェックコードを挿入せずにコンパイル

書式： -fignor-cpu-type

機能： CPU チェックコードを挿入しません。

解説： オプションスイッチに -m68020 または -m68040 を指定した場合に，関数 main () に CPU チェックコードを付加しません。
この場合，**X68000** で実行すると暴走することがあります。ローカルな **X68030** に限定された範囲で使用する以外は，このオプションスイッチは使用しないでください。

-flong-offset　PC 間接命令をすべてロングオフセットにする

書式： -flong-offset

機能： プログラムカウンタ間接命令をすべてロングオフセット (32 ビット) にします。

解説： オプションスイッチに -m68020 または -m68040 を指定した場合に，プログラムカウンタ間接アドレッシングで 32 ビットのロングオフセット使用を強制します。
-fall-bsr スイッチ[9]との併用で，32 ビットフルリロケータブルなコードが作成できます。

9) 「PC 間接形式の使用」(P.11) を参照してください。

-fms-dos　8086 系 C プログラムをコンパイルする

書式： -fms-dos

機能： 8086 系 C プログラムのコンパイルに使用します。

解説： MS-DOS のソースを簡便にコンパイルするために，このオプションスイッチを指定した場合には以下の字句が無視されます。単なる読み飛ばしですので，MS-DOS 上の C 言語では，エラーになる位置にあっても無視されます。

- far
- __far
- near
- __near
- huge
- __huge
- cdecl
- __cdecl
- pascal
- __pascal

-fpic　変数をすべて A5 ベースの間接アドレッシングにする

書式： -fpic

機能： 変数をすべて A5 ベースの間接アドレッシングにします。

解説： 従来の -SX モード[10]で行われる A5 レジスタをベースアドレスとした変数アクセスを行うコード生成を，通常時も行うオプションスイッチです。

しかし，このモードに対応したスタートアップが作成されていませんので，現状ではあまり意味のないオプションスイッチです。

10) 「*X68000 Develop.*」Vol.2 P.43 を参照してください。

1-5 削除された診断メッセージ

次に示す診断メッセージは，頻繁に発生することが多かったことと **GCC** Ver.2.xx ではすでに廃止されていたために，**X680x0 GCC** では思い切って廃止しました。

●プロトタイプは double です
●引き数は double のみマッチします

●プロトタイプは int です
●引き数は int のみマッチします

Chapter 2

X680x0 HAS

本書に収録されているバージョンのHASは，前著「*X68000 Develop.*」収録のHASの機能に加えて，対象とするCPUを68020, 68030, 68040に広げる拡張が行われたものです。本章では，従来のHASから拡張された機能について説明します。

2-1 X680x0 HASの拡張概要

本書に収録されているバージョンの**HAS**では，従来の**HAS**の機能に加えて，対象CPUを広げる拡張を主としたいくつかの機能拡張が行われています。本セクションでは，拡張された機能の概要を説明します。

なお文章中,「X680x0 HASでは」と記載されているのは本書に収録されているバージョンのHASを指し,「従来では」または「X68000 HAS」と記載されているのは前著「*X68000 Develop.*」に収録されているバージョンを指します。

◆対象CPU

X68000 HASでサポートされているCPUは68000および68010でしたが，現在ではそれに加えて68020，68030，68040がサポートされています。68020用**メモリ管理ユニット**68851や，**浮動小数点コプロセッサ**68881，68882のすべての命令も使用可能になりました。

HAS自身は68000の命令のみを使用していますので，**X68000**，**X68030**のどちらでも動作します。

◆浮動小数点実数

浮動小数点コプロセッサのサポートにともない，ソースファイル中で浮動小数点実数がそのまま扱えるようになりました。実数演算ルーチンはアセンブラ内部で用意してありますので，実数の表記にあたっては`FLOATn.X`[1]や浮動小数点コプロセッサは不要です。

また，浮動小数点実数を扱うための**浮動小数点シンボル**も使用できるようになっています。

1) **Human68k**で提供されている浮動小数点演算パッケージですが，浮動小数点コプロセッサで扱えるデータのうち拡張精度実数とパックドデシマルが扱えないため，**HAS**では使用することができませんでした。

◆シンボル数制限の撤廃

従来は，ソースファイル中で使用できるシンボル数の上限をコマンドラインで指定するようになっており，その値にも上限[2]がありましたが，現在ではシンボル数に関する制限はなくなっています。コマンドラインの指定に関係なく，メモリの許す限りのシンボルを使用できるようになっています。

ただし，**オブジェクトファイル**に出力できるシンボル数に上限が存在するため，外部定義・参照シンボルとして使用できるのは32767個までに制限

2) 最大32767個まで指定できました。実際には，予約語もシンボルの一種として扱われているために，もうすこし少なくなります。

されます。

◆ **ソースコードデバッグ機能**

C コンパイラのデバッグオプションで出力される**ソースコードデバッグ情報**に相当するものを，アセンブラソースファイルから作成する機能を追加しました。これによって，SCD.X[3]を利用したアセンブラのソースコードデバッグが可能になりました。

実際には，アセンブラソースファイルから得ることのできる情報が少ないことや， SCD.X 自体の機能的な問題などから実験的機能の域を出ませんが，ある程度の利用価値はあると思います。

3) SHARP の XC v2.0 以降に付属するソースコードデバッガです。本来，C 言語のプログラムをソースコードデバッグするためのものです。

◆ **その他の拡張機能**

その他にも，より小さなオブジェクトを出力するための**最適化**機能の強化，**ローカルラベル**の拡張などいくつかの機能拡張を行っています。

2-2 拡張された文法

X680x0 HASでは，浮動小数点実数への対応などのために，従来のアセンブリ言語の文法に拡張が行われています。本セクションでは，拡張された文法について説明します。

2-2-1 データサイズコードの拡張

従来は，**データサイズ**として`.b`(バイト)，`.w`(ワード)，`.l`(ロングワード)，`.s`(ショート)の4種類を指定することができましたが[1)]，新たに次のデータサイズが追加されています。これらのサイズは浮動小数点コプロセッサ命令で使用できるほか，拡張されたいくつかの疑似命令でもサポートされています。

1) 「*X68000 Develop.*」Vol.1 P.79を参照してください。

- `.s`　単精度実数(4バイト)
- `.d`　倍精度実数(8バイト)
- `.x`　拡張精度実数(12バイト)
- `.p`　パックドデシマル(12バイト)

“`.s`”は従来ショートサイズ指定として扱われていましたが，浮動小数点命令や疑似命令では，単精度実数の指定として扱われることになります。

2-2-2 浮動小数点実数の扱い

新たに浮動小数点コプロセッサ命令をサポートするようになったため，ソースファイル中に直接浮動小数点実数を表記することができるようになっています。

● 浮動小数点定数

浮動小数点実数を定数として表記する場合には，指定したい値をそのまま実数として表記する実数表記と，浮動小数点コプロセッサなどで内部的に扱われる16進数の内部表現表記を使用することができます。

◆実数表記

実数表記では，指定したい値をそのまま実数として表記します。表記した値が浮動小数点実数として扱われるためには，次の条件のいずれかを満たす必要があります。

- **“0f” で始まっている**
- **小数点を含んでいる**
- **指数表現を含んでいる**

List 2 – 1 浮動小数点の実数表記

```
1:          fmove.x #0f100,fp0      * 0f100 = 100.0
2:          .dc.s   100.0
3:  SYMBOL  .fequ   1e+2            * 1e+2 = 100.0
```

◆内部表現表記

内部表現表記では，指定したい値を浮動小数点コプロセッサなどで内部的に扱われる表現で表記します。内部表現では，指定した値をどのように扱うかはデータサイズに依存するので，これを扱う命令や疑似命令では，データサイズを省略することはできません。
内部表現の表記には，次の 2 つの方法があります。

- **“!” の後に 16 進の内部表現を直接表記する**
  16 進数の内部表現を，“!” を先頭に置いて必要な桁数だけ直接表記する。データサイズによって，必要な桁数は Table 2-1 のように変わる
- **整数定数値を内部表現に必要なだけ並べる**
  整数定数値を，必要なデータ数だけ “,” で区切って並べる。1 つの整数定数には 32 ビット (16 進数 8 桁) の値が使用できるので，データサイズによって，必要なデータ数は Table 2-1 のように変わる

Table 2 – 1● 内部表現に必要なデータ数

データサイズ	必要な 16 進の桁数 (桁)	必要なデータ数 (個)
.s (単精度実数)	8	1
.d (倍精度実数)	16	2
.x (拡張精度実数)	24	3
.p (パックドデシマル)	24	3

List 2 – 2 浮動小数点の内部表現表記

```
1:      .dc.d   !3ff0_0000_0000_0000            * 倍精度の +1.0
2:      .dc.x   $3fff0000,$80000000,$00000000   * 拡張精度の +1.0
```

整数定数値による内部表現表記では，その値が 16 進表記であるかどうかのチェックはしていません2)。このために

実数表記と内部表現表記の混同が起きやすい場合がある

ので，注意してください。
たとえば，List 2-3 では，1 行目，2 行目とも浮動小数点実数 1.0 を意図していると考えられますが，後者は整数定数値であるため，実際には，List 2-4 と同じ意味になり，意図したのとまったく違った値として扱われてしまいます。
単精度実数 (.s) 以外のデータサイズでは，整数定数 1 つでは内部表現にならないため，List 2-5 のような誤りはエラーとして検出されます。

2) 数値をその基数で区別するのが文法上不自然なためです。不便なようですが，たとえば "1+1" と "$1+1" の結果が異なるというのは不自然でしょう。

List 2 3 内部表現表記の混同が起きやすい例

```
1:      fmove.s #1.0,fp0        * 単精度実数値 +1.0
2:      fmove.s #1,fp0          * 実数値 1.0 ???
```

List 2 4 実際はこのように扱われる

```
1:      fmove.s #1.0,fp0
2:      fmove.s #$00000001,fp0  * おおよそ +1.40e-45 になる
```

List 2 5 エラーとなる例

```
1:      fmove.d #1,fp0          * 倍精度実数では整数定数が 2 つ必要
```

● 浮動小数点シンボル

浮動小数点値を定義するための，浮動小数点シンボルを使用することができます。浮動小数点シンボルには整数シンボルと同様に，**.fequ** 疑似命令3)によって定義される，その値を変更することのできない**不変シンボル**と，**.fset** 疑似命令4)によって定義され，値をソースファイル中で変更可能な**可変シンボル**があります。

3) 「不変浮動小数点シンボル値の定義」(P.43) を参照してください。

4) 「可変浮動小数点シンボル値の定義」(P.44) を参照してください。

List 2 6 浮動小数点シンボル

```
1:      PI      .fequ   3.14159265
2:      VALUE   .fset   +1.0
3:      VALUE   .fset   -1.0
```

● 浮動小数点式

浮動小数点式では，アセンブラ演算子[5]のうち単項演算子の正負 (+, -), 2 項演算子の四則演算 (+, -, *, /) のみを使用することができます。これらの演算子を使用して，次の値の間での演算を行うことができます。

- **浮動小数点実数 (実数表記に限られます)**
- **浮動小数点シンボル**
- **整数シンボル (定数である必要があります)**

また，シンボルの前方参照はできません。

整数オペランドを必要とする所で浮動小数点式を使用すると，演算結果の小数点以下を切り捨てられた値が使用されます (List 2-7)。

5)「*X68000 Develop.*」Vol.I P.84 を参照してください。

List 2　7　浮動小数点式

```
1:      move.l  #3.1415*1000.0,d0    * 結果の3141.5の小数点以下を
2:                                   * 切り捨てた3141が使用される
3:                                   * move.l #3141,d0 と同じになる
```

2-2-3 数値定数表記の拡張

従来，16 進数，8 進数，2 進数の定数は，それぞれ先頭に “$”, “@”, “%” をつけて示していましたが[6]，これらに加えて 16 進数に “0x”，8 進数に “0o”，2 進数に “0b” という表記が使用できるようになっています (List 2-8)。

これらの表記は **C** 言語での定数表記に類似しています。**C** 言語では 8 進数を数値の先頭に “0” を置くことで表しますが，**X680x0 HAS** では従来との互換のために，この表記を 10 進数として扱いますので注意が必要です。

6)「*X68000 Develop.*」Vol.I P.84 を参照してください。

List 2　8　拡張された数値定数表記

```
1:      .dc.l   0x00006800      * $00006800と同じ
2:      .dc.l   0b1010_1111     * %10101111と同じ
3:      .dc.l   0777            * C 言語では 8 進数になるが,
4:                              * HAS では 10 進数として扱う
```

2-2-4 識別名の拡張

対応 CPU の拡張にともなって，新たなキーワードの追加が行われています。また，アセンブラによって定義されている特殊シンボルにも，いくつかの拡張が行われています。

● シンボル

従来から使用できた「数値シンボル」,「レジスタリストシンボル」,「マクロシンボル」に加えて，新たに「浮動小数点シンボル」が追加されています[7)]。

また，アセンブラによって定義されている特殊シンボルに，いくつかの拡張が行われています。

7) 「浮動小数点シンボル」(P.22) を参照してください。

◆ ローカルラベル

従来のローカルラベル[8)]である “@@:” に加えて，“1:” ～ “9:” の 9 種類のローカルラベルが使用できるようになりました。“@@:” 同様に，これらのシンボルはソースファイル中で何度も使用することができます。

“1f” ～ “9f” で，その位置以降の最も近い対応するローカルラベルを，“1b” ～ “9b” で，その位置以前の最も近い対応するローカルラベルを参照します (Fig. 2-1)。

8) 『*X68000 Develop.*』Vol.I P.83 を参照してください。

```
1: 1:
2:        bra 2f
3:        bra 1b
4: 2:
```

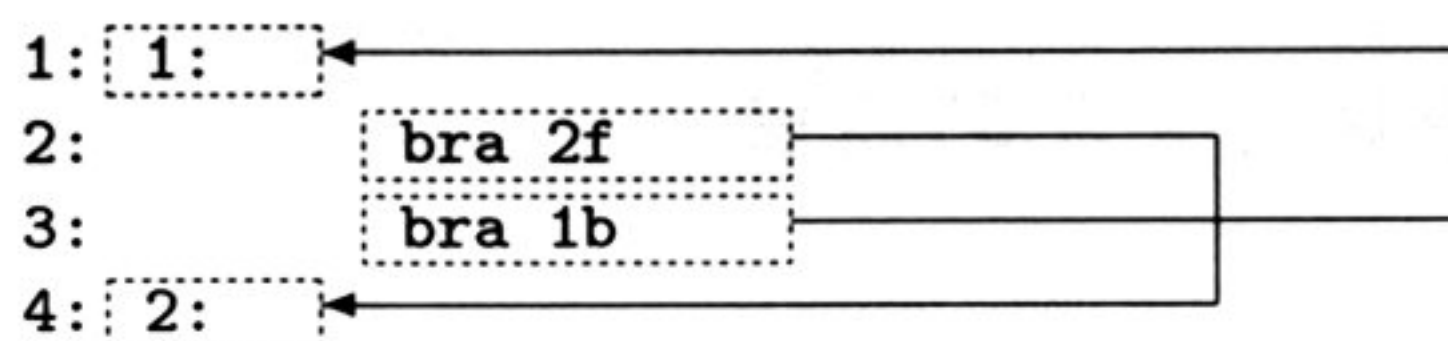



Fig. 2－1● 拡張されたローカルラベル

◆ “$”

従来，現在のロケーションアドレスの値を保持するシンボルとして “*” が使用されていました[9)]が，それに加えて “$” が使用できるようになりました。

“*” は「現在処理している行が始まるロケーションアドレス」を保持するのに対して，“$” は「これからコードを出力しようとするロケーションアドレス」を保持しているという違いがあります (List 2-9)。

9) 『*X68000 Develop.*』Vol.I P.82 を参照してください。

List 2 9 ロケーションアドレスシンボル

```
1:      .org    0
2:      .dc.l   *,*,*    * $00000000 を 3 つ出力する
3:
4:      .org    0
5:      .dc.l   $,$,$    * $00000000,$00000004,$00000008 を出力する
```

● キーワード

X680x0 HAS で新たに対応したCPUによって追加された**ニーモニック**[10]や**レジスタ名**[11]が，**予約語**として追加されています。追加された予約語が有効となるのは，.cpu 疑似命令[12]などで，その予約語を使用するアセンブル対象命令セットが選択された場合に限られます。

List 2-10 では，68020, 68030, 68040で追加された予約語である"MSP"(マスタスタックポインタ) が使用されています。

2 行目と 5 行目は同じ内容ですが，2 行目をアセンブルする際には，1 行目によって命令セットが 68030 になっているので，予約語である"MSP"を数値シンボルとして扱おうとしているためにエラーが発生します。

5 行目をアセンブルする際には，4 行目によって命令セットが 68000 になっているので，68000 には存在しないレジスタである"MSP"は予約語とはならないために，正常にアセンブルすることができます。

10)『X68000 Develop.』Vol.I P.83 を参照してください。

11)『X68000 Develop.』Vol.I P.83 を参照してください。

12)「アセンブル対象命令セットの指定」(P.41) を参照してください。

List 2-10 予約語の追加

```
1:          .cpu    68030
2:  MSP     .equ    1               * エラーになる
3:
4:          .cpu    68000
5:  MSP     .equ    1               * 可能
```

2-2-5 新たな命令と疑似レジスタ

X680x0 HAS では，68000 や 68010 で大きなプログラムをアセンブルするうえでの制限のひとつである，**ブランチ命令**のジャンプ先に 32 ビット**オフセット**が使用できない[13]などの点を補うための新たな命令を追加しています。

13)68000 や 68010 でのブランチ命令のジャンプ先は，そのブランチ命令のある位置から前後 32K バイトずつに限られています。

● ジャンププブランチ命令

ブランチ命令 (BRA, BSR, Bcc) に対応する，ジャンプブランチ命令のニーモニック"JBRA"，"JBSR"，"JBcc"が追加されています。

これらの命令をブランチ命令の代わりに使用すると，ジャンプ先が 16 ビットオフセットで表現できなくなった場合に，これを同等のジャンプ命令で置き換えるようになります。

List 2-11 では，ジャンプブランチ命令 JBSR, JBEQ を使用しています。ジャンプ先である LABEL1 や LABEL2 が 16 ビットオフセットで届く場合は，これらの命令はそれぞれ BSR 命令，BEQ 命令になりますが，16 ビットオフセットの範囲外[14]になると，これらの命令は同等のジャンプ命令に置き換え

14)68000 や 68010 では，エラーになります。

られます。

JBSR 命令は JSR 命令になりますが，BEQ 命令は，1 命令で対応するジャンプ命令がないため[15]，List 2-12 のように，逆条件のブランチ命令 (BNE 命令) + ジャンプ命令 (JMP 命令) という 2 命令に変換されます。

15)68K 系 CPU には条件ジャンプ命令はありません。

List 2 11 ジャンプブランチ命令

```
1:      jbsr    LABEL1
2:      jbeq    LABEL2
```

List 2 12 JBEQ 命令の変換

```
1:      bne     *+8
2:      jmp     LABEL2
```

また，コマンドラインに -b スイッチ[16]を指定することで，ソースリスト中のすべてのブランチ命令をジャンプブランチ命令として扱うようになります。

16)「ロングワードの PC 間接を絶対ロングにする」(P.36) を参照してください。

● 疑似レジスタ OPC

プログラムカウンタ (PC) に対応する疑似レジスタ OPC (optional PC) が追加されています。

このレジスタを PC の代わりに使用すると，**プログラムカウンタ間接形式**[17]のオフセットが 16 ビットで表現できなくなった場合に，同等の**絶対ロングアドレス形式**[18]で置き換えるようになります。

List 2-13 では，シンボル VALUE の値を A0 レジスタに代入しています。VALUE がこの命令から 16 ビットオフセットで届く場合は，この命令は**ディスプレースメントつきプログラムカウンタ間接形式**[19]として扱われますが，16 ビットオフセットの範囲外[20]になると，絶対ロングアドレス形式に置き換えられます (List 2-14)。

この疑似レジスタが使用できるのは，ディスプレースメントつきプログラムカウンタ間接形式に限られます。プログラムカウンタを使用するその他のアドレス形式では，同等な絶対ロングアドレス形式に単純に置き換えることができないため，OPC を使用するとエラーになります。

17)Appendix A「アドレス形式の表記」(P.162) を参照してください。

18)Appendix A「アドレス形式の表記」(P.166) を参照してください。

19)Appendix A「アドレス形式の表記」(P.163) を参照してください。

20)68000 や 68010 では，エラーになります。

List 2 13 疑似レジスタ OPC

```
1:      lea.l   VALUE(opc),a0
```

List 2 14 アドレス形式の変換

```
1:      lea.l   VALUE.l,a0
```

コマンドラインに -b スイッチ[21] を指定すると，ソースリスト中のすべてのディスプレースメントつきプログラムカウンタ間接形式の PC を OPC として扱うようになります。

21)「ロングワードの PC 間接を絶対ロングにする」(P.36) を参照してください。

2-3 その他の拡張機能

本セクションでは，アセンブリ言語の文法以外の拡張機能について説明します。

2-3-1 最適化の強化

X680x0 HASでは，従来より小さなオブジェクトを出力するために，新たに以下の最適化を行うようになっています。

これらの最適化は，コマンドラインに -c スイッチ[1] を指定することで禁止することができます。

1) 「HAS v2.x 互換の最適化を行う」（P.38）を参照してください。

◆絶対ショートアドレス形式への対応

従来では，サイズ指定のない絶対アドレス形式が使用された場合には必ず絶対ロングアドレス形式として扱い，コマンドラインに -a スイッチ (絶対ショートアドレス対応モード[2]) が指定されたときに限って，指定された値が絶対ショートの範囲内 ($00000000 ～ $00007FFF，$FFFF8000 ～ $FFFFFFFF) にある場合に，**絶対ショートアドレス形式**として扱っていました。

2) 「*X68000 Develop.*」Vol.2 P.50 を参照してください。

現在では，サイズ指定をせずに指定された値が絶対ショートの範囲内にあれば，常に絶対ショートアドレス形式として扱うようになっています。つまり，常に -a スイッチが指定されているのと同じように扱うということです。

ただし，値にサイズを明示すると，必ずそのサイズで扱われるようになります。

List 2 15 絶対ショートアドレス形式への対応

```
1:      pea.l    $10000          * 絶対ロングサイズ
2:      pea.l    $1000           * 絶対ショートサイズ
3:      pea.l    $1000.l         * サイズ指定があるので絶対ロングサイズ
```

◆ディスプレースメントつきアドレスレジスタ間接の最適化

ディスプレースメントつきアドレスレジスタ間接形式[3] のディスプレースメントの値が 0 である場合には，それと等価なアドレスレジスタ間接形式への変換を行うようになりました。

ただし，値にサイズを明示した場合には，この最適化は行われません。

3) Appendix A「アドレス形式の表記」(P.161) を参照してください。

List 2 16 ディスプレースメントつきアドレスレジスタ間接の最適化

```
1:      move.l  0(a0),d0        * move.l (a0),d0 に変換される
2:      move.l  0.w(a0),d0      * サイズ指定があるので変換は行われない
```

◆無意味なブランチ命令の削除

ブランチ命令の飛び先が次に実行される命令である場合には，そのブランチ命令には意味がないことになります。

X68000 HAS ではこのような場合にも必ずブランチ命令を出力していましたが，現在は，このようなブランチ命令[4]を削除するようになりました。ただし，ブランチ命令にサイズを明示した場合には，この最適化は行われません。

4) コプロセッサブランチ命令 (cpBcc) を含みます。

List 2 17 無意味なブランチ命令の削除

```
1:          bra     NEXT1           * この命令は出力されない
2:  NEXT1:
3:          bra.w   NEXT2           * サイズ指定があるので削除されない
4:  NEXT2:
```

2-3-2 アドレス境界情報の出力

X680x0 HAS では，.align 疑似命令[5] や .quad 疑似命令[6] によって，アドレスを偶数境界以上の境界にそろえることができるようになっています。ですが，複数のオブジェクトファイルをリンクして実行ファイルを作成する場合，従来のリンカでは，オブジェクトを偶数境界でリンクしてしまうため，これらの指定の意味がなくなってしまいます。

Fig. 2-2 (次ページ) のように，.align 疑似命令を含むプログラムをリンクしたとします。Program B では先頭に“.align 4”という境界指定があるので，このアドレスは 4 の倍数になるはずです[7]。ですが，リンクを行うと，その前に配置される Program A が 254 バイトあるので， Program B の先頭アドレスは，プログラム全体の先頭から 254 バイト目，つまり 4 の倍数ではなくなってしまいました。

5) 「アドレス境界の調整」(P.46) を参照してください。

6) 「アドレスの 4 バイト境界の調整」(P.47) を参照してください。

7) Program B をアセンブルする際には，ロケーションカウンタを 0 から始めるので，確かに 4 の倍数になっています。

この問題を解決するためには，リンカが，リンクするオブジェクトファイルをそれぞれ何バイト境界で始めるべきなのかを知る必要があります。先ほどの例の場合，Program B を 4 バイト境界で始めるということがわかれば，Fig. 2-3 のように，Program A と Program B との間に 2 バイトのパディング (つめもの) を置くことで対処ができるわけです。

シャープ純正アセンブラとリンカ (AS.X，LK.X v3.0) では，このアドレス境界の情報をリンカに対して直接オプションスイッチ (/e スイッチ) で与えるようになっていますが，ユーザが常にオプションスイッチ指定をしなければならないというのは非常に不便です[8)]。

そこで **X680x0 HAS** では，オブジェクトファイルに新たにアドレス境界に関する情報を出力するようにしてあります。このアドレス境界情報の追加されたオブジェクトファイルは，**X680x0 HLK** によってリンクすることで，ユーザが特に意識することなく意図したとおりのアドレス境界が得られるようになります。

追加したアドレス境界情報は，**X680x0 HLK** 以外のリンカを使用した場合に悪影響を与えないように，特殊な外部定義シンボルの一種という形を取っています[9)]。このため，アドレス境界情報をもったオブジェクトファイルは，境界情報が無視されることを除いて，**X680x0 HLK** 以外のリンカでもまったく正常にリンクされるようになっています。

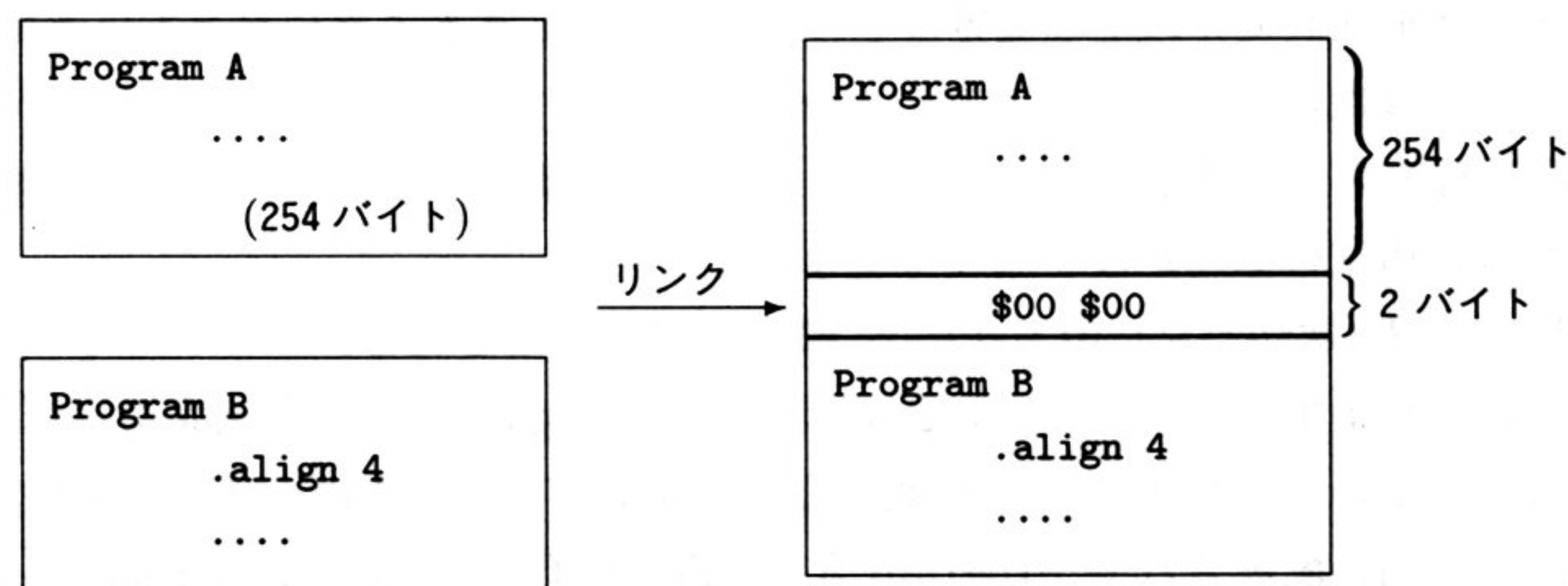
```
Program A                      Program A
    ....                           ....            } 254 バイト
     (254 バイト)     リンク
                      ---->    Program B
Program B                          .align 4
    .align 4                       ....
    ....
```

Fig. 2-2● リンクによってアドレス境界がくずれる例

```
Program A                      Program A
    ....                           ....            } 254 バイト
     (254 バイト)     リンク
                      ---->        $00 $00         } 2 バイト
Program B                      Program B
    .align 4                       .align 4
    ....                           ....
```

Fig. 2-3● 正しくリンクされた例

8) 本来なら，オブジェクトファイルにアドレス境界情報を含めるための拡張を行うべきだと思うのですが・・・。LK.X v1.0 から v2.0 へのバージョンアップでオブジェクトファイルにまったく無意味な変更を行ったにもかかわらず，今回のような場合に必要な拡張を行ってくれないというのには首をかしげてしまいます。

9) 具体的には，"*" ＋ ソースファイル名 ＋ "*" という名前のシンボルが定義されます。**X680x0 HLK** 以外のリンカで，このオブジェクトファイルをリンクしても余計なシンボルが残る以外の影響はありません。

2-3-3 ソースコードデバッグ機能

SHARP 純正の開発キットである **XC** では，v2.0 から **C** 言語プログラムをソースコードデバッグするための機能が用意され，**ソースコードデバッガ** SCD.X によってこの機能を利用することができます。「*X68000 Develop.*」においても，同様に **GCC** と **GDB** によるソースコードデバッグの環境が用意されています。

ですが，これらの機能はアセンブリ言語で書かれたソースプログラムでは利用できず，従来のような逆アセンブルリストによるデバッグに頼るしかありませんでした。そこで，試験的にアセンブリ言語プログラムをソースコードデバッグするための機能を追加しています。

コマンドラインに **-g** スイッチ[10] を指定することで，出力するオブジェクトファイルにソースコードデバッグ情報を追加することができます。

10)「SCD 用デバッグ情報の出力」(P.40) を参照してください。

一般に，プログラムをソースコードデバッグするためには，デバッグするプログラムとそのソースリストとの関係を知るための情報が必要です。**C** 言語におけるソースコードデバッグでは，大まかに次の 3 つの情報を利用しています[11]。

11)「*X68000 Develop.*」Vol.I P.254 を参照してください。

1. プログラムとソースリストの行との対応
2. プログラムのブロック構造
3. プログラムの使用する変数の型と構造

C 言語の場合，コンパイラはソースリストをコンパイルする際にこれらの情報を知ることができます。ところが，アセンブリ言語は構造のない言語ですから，**1.** 以外の情報を知ることはほぼ不可能です。このため，残念ながらデバッグ機能はかなり制限されてしまいます。特に，**GDB** で提供されている強力なデバッグ支援機能はほとんど使うことができず，ほぼ SCD.X でのマウスを使ったデバッグ専用の機能になってしまっています。

2-4 拡張されたオプションスイッチ

X680x0 HASでは，拡張された機能をサポートするなどの目的で，いくつかのオプションスイッチの内容が変更されています。

2-4-1 廃止されたオプションスイッチ

X680x0 HASでは，以下のオプションスイッチが廃止されています。従来との互換のために，廃止されたオプションスイッチを使用しても無視されるだけでエラーなどは発生しませんが，**将来の仕様変更によって完全に廃止される可能性があります。**

-a，-q スイッチは，**v2.x** 互換の最適化を行う場合[1]にのみ使用できます。

- **-a**　絶対ショートアドレス形式対応モードの指定[2]

  X68000 HASでは，このオプションスイッチがない場合には，サイズ指定のない絶対アドレス指定は絶対ロングアドレス形式[3]として扱われていました。

  現在では，常にこのオプションスイッチが指定されているのと同じ状態でアセンブルが行われるようになっているため，このオプションスイッチは廃止されています。

- **-q**　クイックイミディエイト形式への変換禁止[4]

  X68000 HASでは，命令の**クイックイミディエイト形式**への変換が可能な場合，その変換を禁止するためにこのオプションスイッチを使用していました。

  このオプションスイッチ指定では，ソースファイル中の変換可能なすべての命令の変換が禁止されていましたが，現在では，イミディエイト値にサイズを指定することで，命令ごとに個別に変換を禁止することができるようになっています (List 2-18)。そのため，このオプションスイッチは廃止されています。

1) 「HAS v2.x 互換の最適化を行う」(P.38) を参照してください。

2) 「*X68000 Develop.*」Vol.2 P.50 を参照してください。

3) Appendix A「アドレス形式の表記」(P.166) を参照してください。

4) 「*X68000 Develop.*」Vol.2 P.59 を参照してください。

List 2　18　クイックイミディエイトへの対応

```
1:      move.l  #0,d0           * moveq.l #0,d0 に変換される
2:      move.l  #0.1,d0         * 変換されない
```

● -r　　相対セクション命令の使用許可[5)]

● -z　　**HAS** オリジナル機能へのワーニング禁止[6)]

従来は，**SHARP** 純正アセンブラ AS.X との仕様の違いを意識してこれらのオプションスイッチを用意していましたが，**X680x0 HAS** ではこれらのオプションスイッチは廃止されています。現在では，常に実装されているすべての機能が使用できます。

● -m　　最大シンボル数の指定[7)]

X680x0 HAS ではシンボル数の上限がなくなったので，このオプションスイッチは廃止されています。

かわりに，-m スイッチにはアセンブル時の CPU 指定の機能が追加されましたが，指定できる値の範囲が異なるため，従来のコマンドラインがエラーになることはありません。-m スイッチのパラメータが 1000 ～ 32767 の場合には，従来の最大シンボル数指定としてこれを無視しますが，パラメータが 68000，68010，68020，68030，68040 の場合は，CPU 指定として扱います。

5) 「*X68000 Develop.*」Vol.2 P.60 を参照してください。

6) 「*X68000 Develop.*」Vol.2 P.67 を参照してください。

7) 「*X68000 Develop.*」Vol.2 P.55 を参照してください。

2-4-2 変更されたオプションスイッチ

X680x0 HAS では，次のオプションスイッチが変更されています。

● -m　　アセンブル対象命令セットの指定

-m　アセンブル対象命令セットの指定

書式：　-m68000 ··· 68000 命令セットの指定
　　　　-m68010 ··· 68010 命令セットの指定
　　　　-m68020 ··· 68020 命令セットの指定
　　　　-m68030 ··· 68030 命令セットの指定
　　　　-m68040 ··· 68040 命令セットの指定

機能：　アセンブルの対象となる命令セットを指定します。

解説：　アセンブルの対象となる命令セットを指定します。命令セットの指定は，ソースファイル中の .cpu 疑似命令[8)] によるものが最も優先しますが，その指定がない場合には -m スイッチによる指定

8) 「アセンブル対象命令セットの指定」(P.41) を参照してください。

が有効となります。

従来，-m スイッチは最大シンボル数の指定のために使用されていましたが[9]，**X680x0 HAS** ではシンボル数の制限がなくなったため，この機能は削除されています。

9)「*X68000 Develop.*」Vol.2 P.55 を参照してください。

例： ソースファイル source.s を，68030 命令セットを指定してアセンブルします。

```
A>has -m68030 source.s
```

2-4-3 追加されたオプションスイッチ

X680x0 HAS には，対象 CPU の変更によって以下のオプションスイッチが追加されています。

- -b　ロングワードの PC 間接を絶対ロングにする
- -c　**HAS** v2.x 互換の最適化を行う
- -e　外部参照オフセットのデフォルトをロングワードにする
- -g　SCD 用デバッグ情報の出力

-e　外部参照オフセットのデフォルトをロングワードにする

書式： -e

機能： 外部参照オフセットのデフォルトサイズをロングワードにします。

解説： 68020 以降の命令セットを指定してアセンブルを行う場合に，サイズ指定なしに外部参照値が使用されたときのサイズをロングワードにします。命令セット指定が 68000，68010 だった場合には，このオプションスイッチの指定は無視されます。

68020 以降の CPU では，68000，68010 から**アドレス形式**[10]に大幅な拡張が行われています。これによって，従来 16 ビット (ワード) の値しか認められていなかったディスプレースメントつきアドレスレジスタ/プログラムカウンタ間接形式や 8 ビット (ショート) の値しか認められていなかったインデックスつきアドレスレジスタ/プログラムカウンタ間接形式などに，32 ビット (ロングワード) の値が指定できるようになるなど，アドレス形式の自由度が増しています。

10) Appendix A「アドレス形式の表記」(P.160) を参照してください。

これらのアドレス形式が使用されると，アセンブラはオフセット(ディスプレースメント)に指定された値からできるだけサイズが小さくなるような選択を行いますが，オフセットに使用された値に外部参照値が含まれていると，アセンブラだけではその値が求められないためにサイズを決定することができなくなります。**HAS** ではこのような場合，68000，68010 と同じ結果が得られるように，それぞれワード/ショートサイズを選択しますが，-e スイッチを使用することで，ロングワードサイズを選択するようになります。

また 68000，68010 では，ブランチ命令 (BRA，Bcc，BSR) のオフセットには 8 ビット (ショートサイズ) または 16 ビット (ワードサイズ) の値を指定することができましたが， 68020 以降ではさらに 32 ビット (ロングワードサイズ) の値を指定できるような拡張がなされています。これらの命令のジャンプ先が外部参照値であった場合にはワードサイズを選択しますが，-e スイッチを使用することで，これも同様にロングワードサイズを選択するようになります。

ただし，命令やアドレス形式中で明示的にサイズ指定を行った場合には，必ず指定されたサイズを選択します。

このオプションスイッチを指定することで，大きなプログラムを作成する場合に，オフセットが届かないためにリンカでエラーになることはなくなりますが，生成される実行ファイルは 68020 以降専用となります。

例： List 2-19 のソースファイル source.s を，次のように -e スイッチをつけてアセンブルします。

```
A>has -e source.s
```

シンボル LABEL は外部参照シンボルなので，3 行目の BSR 命令はロングワードオフセットでアセンブルされます (List 2-20)。
同じソースファイルを -e スイッチをつけずにアセンブルすると，BSR 命令はワードオフセットになります (List 2-21)。
4 行目の BRA 命令はワードサイズ指定があるため，-e スイッチの有無に関係なく必ずワードオフセットでアセンブルされます。

List 2 19 source.s

```
1:      .cpu    68030
2:      .xref   LABEL
3:      bsr     LABEL
4:      bra.w   LABEL
```

List 2 20 "-e" スイッチを指定した場合

```
1:  00000000                          .cpu    68030
2:  00000000                          .xref   LABEL
3:  00000000 61FF????????             bsr     LABEL
4:  00000006 6000????                 bra.w   LABEL
```

List 2 21 "-e" スイッチを指定しない場合

```
1:  00000000                          .cpu    68030
2:  00000000                          .xref   LABEL
3:  00000000 6100????                 bsr     LABEL
4:  00000004 6000????                 bra.w   LABEL
```

-b　ロングワードのPC間接を絶対ロングにする

書式： -b

機能： ロングワードオフセットのプログラムカウンタ間接形式[11]やブランチ命令を，絶対ロングアドレス形式[12]へ変換します。

解説： 実効アドレスのプログラムカウンタ間接形式やブランチ命令のオフセット（ディスプレースメント）が32ビット（ロングワード）になる場合に，絶対ロングアドレス形式に変換可能なものについて変換を行います。

68000，68010では，実効アドレスのプログラムカウンタ間接形式やブランチ命令のオフセットに32ビットの値を扱うことができないので，大きなプログラムでブランチ命令のジャンプ先が16ビットオフセットで表現できない場合などに“**illegal relative error**”[13]が起きていました。-bスイッチを使用することで，16ビットオフセットで表現できない場合には絶対ロングアドレス形式への変換を行うようになります。

11) Appendix A「アドレス形式の表記」(P.162) を参照してください。

12) Appendix A「アドレス形式の表記」(P.166) を参照してください。

13)「*X68000 Develop.*」Vol.2 P.173 を参照してください。

以下のアドレス形式や命令が，-b スイッチによる変換の対象となります。

◆ディスプレースメントつきプログラムカウンタ間接形式
((adr,PC))
オフセットの値が 16 ビットで表現できなくなると，絶対ロングアドレス形式 (adr.l) に変換する

◆無条件ブランチ命令 (BRA，BSR)
オフセットの値が 16 ビットで表現できなくなると，ジャンプ命令 (JMP，JSR) に変換する

◆条件ブランチ命令 (Bcc)
オフセットの値が 16 ビットで表現できなくなると，反対条件のブランチ命令 + ジャンプ命令 (JMP) の 2 命令に変換する

これらの変換は，ソースファイル中で明示することもできます。プログラムカウンタ間接形式のレジスタ名に疑似レジスタ OPC，ブランチ命令の代わりにジャンプブランチ命令 JBRA，JBSR，JBcc を使用することで，-b スイッチの有無に関係なく，必要に応じて絶対ロングアドレス形式への変換を行うようになります[14]。

ただし，**リロケータブル**なプログラムを作成する場合は，-b スイッチによる変換が行われると出力されるオブジェクトファイルがリロケータブルではなくなってしまうので，このスイッチは指定しないようにしてください。

例： List 2-22 のソースファイル source.s を，次のように -b スイッチをつけてアセンブルします。

```
A>has -b source.s
```

2 行目の BRA 命令のジャンプ先は 16 ビットオフセットでは届かないため，代わりに JMP 命令の命令コード $4EF9 が出力されます (List 2-23)。
同じソースファイルを -b スイッチをつけずにアセンブルすると，BSR 命令は，そのまま 32 ビットオフセットで出力されます[15] (List 2-24)。

14)「新たな命令と疑似レジスタ」(P.25) を参照してください。

15) このプログラムは 68000 や 68010 では実行できません。

List 2　22　source.s

```
1:             .cpu     68030
2:             bra      LABEL
3:             .ds.b    32768
4:   LABEL:
```

List 2　23　"-b" スイッチを指定した場合

```
1:  00000000                                .cpu     68030
2:  00000000 4EF9(01)00008006               bra      LABEL
3:  00000006                                .ds.b    32768
4:  00008006                       LABEL:
```

List 2　24　"-b" スイッチを指定しない場合

```
1:  00000000                                .cpu     68030
2:  00000000 60FF00008004_000               bra      LABEL
3:           08006
4:  00000006                                .ds.b    32768
5:  00008006                       LABEL:
```

-c　HAS v2.x 互換の最適化を行う

書式：　-c

機能：　HAS v2.x 互換の最適化を行います。

解説：　X680x0 HAS で新たに追加された最適化機能を禁止して，従来と同じオブジェクトファイルが得られるようにします。

X680x0 HAS では，従来に比べてよりサイズの小さく高速な命令を出力するために，新たな最適化を行うようになっています[16)]。このために，命令の自己書き換えを行うような一部のプログラムが正常に動作しなくなることがあります。-c スイッチを指定すると，これらの新たな最適化を行わなくなるので，X68000 HAS と同じオブジェクトファイルが得られるようになります。

このオプションスイッチは，従来のバージョン用に書かれたソースファイルに対する互換のために用意されているもので，**将来廃止される可能性があります。**新たに追加された最適化は，命令やそのオペランドにサイズを明示することで抑制することができる

16)「最適化の強化」(P.28) を参照してください。

ので，必要ならば積極的にサイズ指定をすることで，このオプションスイッチに依存しないようなソースにしたほうが安全です。

例： List 2-25 のソースファイルは，新たに追加された最適化によって問題の生じる可能性のある例です[17]。

このソースファイルを **X68000 HAS** でアセンブルすると，List 2-26 のような結果が得られますが，**X680x0 HAS** でアセンブルすると，List 2-27 のようになってしまい，1 行目の MOVE 命令によって書き込んだ D0 の値が次の命令を破壊してしまいます。

このような場合は，-c スイッチをつけてアセンブルすることで，List 2-26 のような結果が得られるようになります。

また，このソースファイルは List 2-28 のようにサイズ指定を追加することで，どちらの **HAS** でも同じ結果を得られるようにすることができます。

17)命令の自己書き換えを行っているので，あまり行儀のよいプログラムとはいえませんが・・・。特に 68030 など，命令キャッシュを備えている CPU での実行を考えると，このような記述は避けた方がいいということはいうまでもないでしょう。

List 2　25　バージョンの違いによって結果に違いの出るソースリスト

```
1:          move.w  d0,POSITION+2
2:          ...
3:          ...
4: POSITION:
5:          move.l  0(a0),d1          * v3.0 だと最適化によって
6:                                    * move.l (a0),d1 に変更される
```

List 2　26　X68000 HAS でのアセンブル例

```
1: 00000000 33C0(01)00000008         move.w  d0,POSITION+2
2: 00000006                          ...
3: 00000006                          ...
4: 00000006                 POSITION:
5: 00000006 22280000                 move.l  0(a0),d1
6: 0000000A
```

List 2　27　X680x0 HAS でのアセンブル例

```
1: 00000000 33C0(01)00000008         move.w  d0,POSITION+2
2: 00000006                          ...
3: 00000006                          ...
4: 00000006                 POSITION:
5: 00000006 2210                     move.l  0(a0),d1
6: 00000008
```

List 2 28 バージョンに依存しないための書き換え

```
1:          move.w  d0,POSITION+2
2:          ...
3:          ...
4:  POSITION:
5:          move.l  0.w(a0),d1      * サイズ指定があるので
6:                                  * move.l (a0),d1 への変更は行われない
```

-g SCD用デバッグ情報の出力

書式： -g

機能： オブジェクトファイルに，SCD用デバッグ情報を出力します。

解説： SCD.Xを使用してアセンブラソースのソースコードデバッグを行うために，オブジェクトファイルにソースコードデバッグ情報を出力します[18]。

このオプションスイッチを指定すると，アセンブラがソースファイルから自動的にデバッグ情報を出力するため，デバッグ情報を明示するためのソースコードデバッグ疑似命令はすべて無視されるようになります。ですから，C言語のプログラムをGCCで-gスイッチ[19]をつけてコンパイルした際の出力アセンブラソースファイルのように，始めからソースコードデバッグ情報のついているソースファイルをアセンブルするときには，このオプションスイッチは使用しないでください。

アセンブラソースファイルから得ることのできるデバッグ情報は非常に少ないため[20]，このオプションスイッチで得られるのは，実際にはソースリストの行番号とオブジェクトの対応程度になります。また，SCD.X自体の機能的な問題のため[21]，C言語でのソースコードデバッグに比べて，デバッグ機能はかなり制限されます。

例： ソースファイル source.s を，ソースコードデバッグ情報の出力を指定してアセンブルします。

```
A>has -g source.s
```

18)「ソースコードデバッグ機能」(P.31) を参照してください。

19)「*X68000 Develop.*」Vol.2 P.14 を参照してください。

20) アセンブリ言語の性質上当然のことなのですが。

21) もともとこのような使い方を想定していないので，ある程度仕方がないことはいえます。

2-5 拡張されたアセンブラ疑似命令

X680x0 HAS では，対応 CPU の拡張や浮動小数点実数への対応のために，いくつかの疑似命令が追加/拡張されています。

2-5-1 アセンブラ制御

対応 CPU が拡張されたため，CPU 指定に関する疑似命令が追加/拡張されました。

- `.cpu`　アセンブル対象命令セットの指定
- `.fpid`　浮動小数点コプロセッサ ID の指定

.cpu　アセンブル対象命令セットの指定

書式：　`.cpu 68000` ･･･ 68000 命令セットの指定
`.cpu 68010` ･･･ 68010 命令セットの指定
`.cpu 68020` ･･･ 68020 命令セットの指定
`.cpu 68030` ･･･ 68030 命令セットの指定
`.cpu 68040` ･･･ 68040 命令セットの指定

機能：　アセンブルの対象となる命令セットを指定します[1)]。

解説：　アセンブルの対象となる命令セットを指定します。**X68000 HAS** では，68000 と 68010 のいずれかのみが指定可能でしたが，現在ではこれらに加えて 68020，68030，68040 の指定ができるようになっています。

68020，68030 のいずれかが指定された場合には，同時に浮動小数点コプロセッサ 68881，68882 の命令セットも有効となります。また，68020 が指定された場合には，同様にメモリ管理ユニット 68851 の命令セットも有効となります。

`.cpu` 疑似命令による指定がない場合のデフォルトの命令セットは，コマンドライン上での `-m` スイッチ[2)] による CPU 指定があ

1) 「*X68000 Develop*」Vol.I P.100 を参照してください。

2) 「アセンブル対象命令セットの指定」（P.33）を参照してください。

ればその指定に従います。コマンドラインによる指定もない場合は，アセンブル時に **HAS** が **X68000** 上で動作している場合には **68000**，**X68030** 上で動作している場合には **68030** になります。

List 2 29 .cpu 疑似命令

```
1:  * 68020 命令セットを指定して，68020 専用の命令である
2:  * CALLM, RTM命令をアセンブルする
3:          .cpu    68020
4:          callm   #4,(a2)
5:          rtm     a1
```

.fpid 浮動小数点コプロセッサ ID の指定

書式： .fpid <式>

機能： 浮動小数点コプロセッサ命令に使用する**コプロセッサ ID** を指定します。

解説： 対応 CPU の拡張によって，CPU が浮動小数点コプロセッサをサポートするようになったために追加された疑似命令です。

浮動小数点コプロセッサ命令に使用するコプロセッサ ID を指定します。<式> の値は 0 ～ 7 の定数でなければなりません。.fpid 疑似命令による指定がない場合には，デフォルトとして 1 が使用されます。

68000 系 CPU のコプロセッサ命令では，命令中の ID フィールドに指定する 0 ～ 7 のコプロセッサ ID によって，対象となるコプロセッサを選択します[3)]。
CPU 内蔵のコプロセッサでは，どの ID がどのコプロセッサを指すのかは固定されていますが[4)]，外付けのコプロセッサの場合は，ID とコプロセッサの対応は，CPU とコプロセッサの間のハードウェアによって決まります。**X680x0 HAS** で使用できるコプロセッサでは，浮動小数点コプロセッサ 68881，68882 がそれにあたります。.fpid 疑似命令では，この浮動小数点コプロセッサ命令が使用されたときに，ID フィールドにどの ID を使用するかを指定します。

X68030 では，浮動小数点コプロセッサには ID 1 を使用するようになっています。.fpid 疑似命令による指定がない場合のデフォ

3) 68000 と 68010 にはコプロセッサインタフェイスがないため，この記述は当てはまりません。これらの CPU でコプロセッサ命令を実行したり，これら以外の CPU でも指定した ID に対応するコプロセッサのないコプロセッサ命令を実行したりすると，F ライントラップという例外が発生します。**Human68k** ではこの例外を DOS コールや FLOATn.X の呼び出しに用いています。

4) 68030 内蔵メモリ管理ユニット (MMU) の ID は 0，68040 内蔵 MMU の ID は 2，68040 内蔵浮動小数点コプロセッサの ID は 1 となっています。また，68020 用 MMU 68851 は，外付けコプロセッサですが，必ず ID 0 で使用することになっているようです。

ルトはID 1になっていますし，IDに1以外の値を指定すると浮動小数点コプロセッサ命令が実行できなくなってしまいますので，通常はこの疑似命令によってIDを指定する必要はありません。

List 2 30 .fpid 疑似命令

```
1:  * 以後の浮動小数点コプロセッサ命令の ID を 4 とする
2:          .fpid   4
```

2-5-2 シンボルの定義

浮動小数点シンボル[5]が追加されたため，その定義のための疑似命令が追加されました。浮動小数点シンボルはアセンブル命令セットに関係なく使用できますので，これらの疑似命令も常に使用可能です。

5) 「浮動小数点シンボル」(P.22) を参照してください。

- ●.fequ　不変浮動小数点シンボル値の定義
- ●.fset　可変浮動小数点シンボル値の定義

.fequ 不変浮動小数点シンボル値の定義

書式： <シンボル> .fequ[.<サイズ>] <浮動小数点式>

機能： 不変浮動小数点シンボルの値を定義します。

解説： 新たに浮動小数点シンボルが使用できるようになったため追加された疑似命令です。

オペランドフィールドの **<浮動小数点式>** の値を，ラベルフィールドの **<シンボル>** に割り当てます。浮動小数点シンボルは前方参照ができないため，この定義はそれが行われた位置からソースの最後まで有効になります。また，シンボルの値をほかの値に再定義することはできません。

<浮動小数点式> に実数表記[6]を使用する場合には **<サイズ>**の指定が省略できますが，内部表現表記[7]を使用する場合にはデータサイズによってその表現が変わるため，**<サイズ>** 指定を省略することはできません。

<サイズ> に指定できるのは .s (単精度実数)，.d (倍精度実数)，.x (拡張精度実数)，.p (パックドデシマル) のいずれかです。

6) 「実数表記」(P.21) を参照してください。

7) 「内部表現表記」(P.21) を参照してください。

List 2　31　.fequ 疑似命令

```
1:  * 浮動小数シンボル PI に値を定義する
2:  PI       .fequ   3.14159265
3:
4:  * 浮動小数シンボル FLOAT1 に倍精度実数値 1.0 を定義する
5:  * 内部表現なのでサイズが必要となる
6:  FLOAT1   .fequ.d !3ff00000_00000000
```

.fset　可変浮動小数点シンボル値の定義

書式：　**<シンボル> .fset[.<サイズ>] <浮動小数点式>**

機能：　浮動小数点シンボルに一時的な値を割り当てます。

解説：　新たに浮動小数点シンボルが使用できるようになったため追加された疑似命令です。

オペランドフィールドの **<浮動小数点式>** の値を，ラベルフィールドの **<シンボル>** に一時的に割り当てます。**.equ** 疑似命令[8] に対する **.set** 疑似命令[9] と同様に，値の定義をソースファイル中で何度でも変更できます。ただし，すでに **.set** 疑似命令で定義された数値シンボルに，**.fset** 疑似命令によって浮動小数点値を割り当てることはできません。

.fequ 疑似命令と同様に，**<浮動小数点式>** に実数表記を使用する場合には **<サイズ>** の指定が省略できますが，内部表現表記を使用する場合にはデータサイズによってその表現が変わるため **<サイズ>** 指定を省略することはできません。

8)「*X68000 Develop.*」Vol.I P.108 を参照してください。

9)「*X68000 Develop.*」Vol.I P.108 を参照してください。

List 2　32　.fset 疑似命令

```
1:  FLOAT    .fset   0.1
2:           ...              シンボル FLOAT の値は 0.1
3:           ...
4:  FLOAT    .fset   0.2
5:           ...              ここから先は，シンボル FLOAT の値は 0.2 になる
6:           ...
7:
8:  INTEGER .set     0
9:
10: INTEGER .fset    0.3      INTEGER は整数シンボルなので，エラーになる
```

2-5-3 データの定義

浮動小数点実数値が扱えるようになったため，それに合わせてデータ定義に関する疑似命令が拡張されました。また，対応 CPU の拡張により，偶数境界以外のアドレス境界への調整が必要になったため，そのための疑似命令が追加されています。

- .dc　　定数データの定義
- .dcb　　定数ブロックの定義
- .ds　　メモリ領域の確保
- .align　　アドレス境界の調整
- .quad　　アドレス境界の調整

.dc　定数データの定義

書式： .dc[.<サイズ>]　<式>[,<式> …]

機能： 定数データを定義します[10]。

解説： 浮動小数点実数に対応したため，**<サイズ>** に .s (単精度実数), .d (倍精度実数), .x (拡張精度実数), .p (パックドデシマル) が指定できるようになりました。

10)「*X68000 Develop.*」Vol.1 P.114 を参照してください。

List 2　33　.dc 疑似命令

```
1:  * パックドデシマル定数データ 1.0, 2.0, 3.0 を定義する
2:          .dc.p   1.0,2.0,3.0
```

.dcb　定数ブロックの定義

書式： .dcb[.<サイズ>]　<長さ>, <式>

機能： 定数ブロックを定義します[11]。

解説： 浮動小数点実数に対応したため，**<サイズ>**に .s (単精度実数), .d (倍精度実数), .x (拡張精度実数), .p (パックドデシマル) が指定できるようになりました。

11)「*X68000 Develop.*」Vol.1 P.115 を参照してください。

List 2　34　.dcb 疑似命令

```
1:  * 拡張精度実数値 1.4142 を 10 個分定義する
2:          .dcb.x  10,1.4142
```

.ds　メモリ領域の確保

書式：　.ds[.<サイズ>]　<長さ>

機能：　メモリ領域を確保します[12]。

解説：　浮動小数点実数に対応したため，**<サイズ>** に .s (単精度実数), .d (倍精度実数), .x (拡張精度実数), .p (パックドデシマル) が指定できるようになりました。

12)「*X68000 Develop.*」Vol.1 P.115 (マニュアル版では P.116) を参照してください。

List 2　35　.ds 疑似命令

```
1:  * 倍精度実数を 16 個分格納できる領域を確保する
2:          .ds.d   16
```

.align　アドレス境界の調整

書式：　.align **<境界値>**[, **<式>**]

機能：　アドレス境界の調整を行います。

解説：　対応 CPU の拡張によって追加された疑似命令です。

次の命令のアドレスが指定した **<境界値>** の倍数になるように，ロケーションカウンタを進めます。**<境界値>** には，2～256 の 2 の累乗を定数で指定しなければなりません。ですが，**Human68k** の実行ファイルは 16 の倍数のアドレスにロードされるようになっているので，通常の実行ファイルを作成するうえでは 16 以上の境界値指定を行っても意味がありません。

<式>には，アドレス境界を合わせるために出力するデータを指定します。指定できるのは，$0000 ～ $ffff の間の定数に限られます。 **<式>** の指定が省略された場合には，現在のセクション[13]がテキストセクションであった場合には **$4e71** (nop の命令コード)，それ以外の場合には **$0000** が使用されます。

境界の調整は次のように行われます。まず，アドレスを偶数にす

13)「*X68000 Develop.*」Vol.1 P.87 を参照してください。

るために，**.even** 疑似命令[14] と同等の処理が行われます。その後，アドレスが指定した境界に達するまで，**<式>** の値がワードで出力されます。

14)「*X68000 Develop.*」Vol.1 P.116 を参照してください。

List 2 36 .align 疑似命令

```
1: * ロケーションカウンタを 16 の倍数に進める
2:         .align  16
```

.quad アドレスの 4 バイト境界の調整

書式： .quad

機能： アドレスの 4 バイト境界の調整を行います。

解説： 対応 CPU の拡張によって追加された疑似命令です。

次の命令のアドレスが 4 の倍数になるように，ロケーションカウンタを進めます。**.align 4** とまったく同じ処理が行われます。

List 2 37 .quad 疑似命令

```
1: * ロケーションカウンタを 4 の倍数に進める
2:         .quad
```

2-6 変更された診断メッセージ

X680x0 HASでは，機能拡張にともなって，エラーメッセージやワーニングメッセージの一部が変更されています。

2-6-1 エラーメッセージ

最大シンボル数の制限がなくなったため，エラーメッセージ“`Abort:  Too many symbols`”[1] は次のメッセージに変更されています。

- **`Abort:  Too many external symbols`**
  ソースファイル中で定義/参照された外部シンボル数の合計が 32767 個を超えました。
  オブジェクトファイルのフォーマット上，これ以上の外部シンボルを扱うことはできませんので，このエラーが出力された場合の対策は基本的にはありません[2]。

2-6-2 ワーニングメッセージ

常に**HAS**オリジナルの機能を認めるようになったため，レベル 2 ワーニングメッセージ“`Warning:  HAS expanded specifications`”[3] は廃止されました。また，次のワーニングが追加されています。

◆レベル 1 ワーニング

・`Warning:  index size not specified`
インデックスレジスタとして使用しているレジスタのサイズ指定が省略されています。
サイズの省略されたインデックスレジスタはワードサイズとして扱われますが，まぎらわしい表記を避けるため，本来なら明記すべき所です。

1) 「*X68000 Develop.*」Vol.2 P.172 を参照してください。

2) このエラーが出るほどの巨大なソースファイルだと，おそらくアセンブル以外の作業にも支障をきたすはずなので，まず心配は不要だと思いますが。

3) 「*X68000 Develop.*」Vol.2 P.174 を参照してください。

2-7 AS.X v3.0 との非互換性

X680x0 HAS は，純正アセンブラ AS.X v3.0 に対して上位コンパチブルの機能をもつように作られてはいますが，仕様のごく一部に非互換である部分が存在します。そのなかには，不合理であると思われる仕様をあえて採用しなかったものもありますが，本セクションでは現在わかっている非互換性について説明します[1)]。

1) ほかにも，ドキュメントとして明記されている以外の部分で，互換性のない仕様が存在するかもしれません。もっとも，この文章を執筆している時点では AS.X v3.0 にはかなりのバグが残っているようで，バグとも仕様ともつかない違いが数多くあることは事実なのですが・・・。

2-7-1 自動アラインメント

AS.X では，命令やワード以上のサイズのデータを出力するとき，ロケーションカウンタの値が奇数であれば，最初に 1 バイトの **$00** を出力して，ロケーションカウンタを偶数にしてから命令やデータを出力します。

一方 **X680x0 HAS** では，このような自動アラインメントは行わず，代わりにワーニング“`illegal alignment`”[2)] を出力してから，命令やデータをそのまま出力します。

2) 「*X68000 Develop.*」Vol.2 P.174 を参照してください。

List 2-38 のソースファイルを AS.X でアセンブルすると，3 行目のデータ出力の前に自動アラインメントが行われ，List 2-39 のような結果となります。一方，**X680x0 HAS** でアセンブルすると，奇数アドレスへのワードデータ出力となる 3 行目と 4 行目でワーニングメッセージが出力され，List 2-40 のような結果が得られます。

List 2 38 自動アラインメントの有無

```
1:          .dc.b   1              * ここでアドレスが奇数になる
2:  DATA:
3:          .dc.w   1
4:          .dc.w   2
```

List 2 39 AS.X でのアセンブル例

```
1:  00000000 01                        .dc.b   1
2:  00000001                 DATA:
3:  00000001 000001                    .dc.w   1  * 自動アラインメント
4:  00000004 0002                      .dc.w   2
```

List 2 40 X680x0 HAS でのアセンブル例

```
1:  00000000 01                        .dc.b   1
2:  00000001                 DATA:
3:  00000001 0001                      .dc.w   1  * この 2 行に対して
4:  00000003 0002                      .dc.w   2  * ワーニングが出力される
```

2-7-2 浮動小数点式の扱い

3)「浮動小数点式」(P.23) を参照してください。

整数オペランドを必要とするところに浮動小数点式3) が表記された場合，AS.X v3.0 では演算前の各数値の小数点以下を切り捨ててから演算を行いますが，**X680x0 HAS** では全体の浮動小数点演算を行った後で小数点以下を切り捨てます (List 2-41)。

また，AS.X と **X680x0 HAS** では同じ浮動小数点実数を与えた際に，誤差の関係で異なる内部表現が得られることがあります。

List 2 41 浮動小数点式の扱い

```
1:     .dc.l   1.5*10.0        * AS.Xでは 1 * 10 なので 10 になる
2:                             * HAS.Xでは演算結果 15.0 を切り捨てて
3:                             * 15 になる
```

2-7-3 .align 疑似命令

4)「アドレス境界の調整」(P.46) を参照してください。

アドレス境界を指定した場合，境界合わせのためにデータを出力する際の規則が AS.X v3.0 と **X680x0 HAS** とで異なっています4)。

AS.X v3.0 では，境界合わせのために式で指定されたデータはバイト単位で扱われます。ワード単位で扱うためには，式にサイズ (.w) を指定する必要があります。また，式の指定を省略した場合には，値として $00 が使われます。

一方 **X680x0 HAS** では，境界合わせに先立って必ず偶数境界が合わせられます。その後，式で指定した値をワード単位として出力します。式の指定を省略すると，現在のセクションがテキストセクションである場合には $4e71 (nop の命令コード)，テキストセクション以外である場合には $0000 が使われます。

List 2-42 のソースファイルを AS.X でアセンブルすると，境界合わせのために出力されるデータは List 2-43 のようになります。一方，同じソースファイルを **X680x0 HAS** でアセンブルすると，List 2-44 のような結果が得られます。

List 2　42　.align 疑似命令の違い

```
 1:     .text
 2:     .org    3
 3:     .align  8
 4:
 5:     .data
 6:     .org    3
 7:     .align  8
 8:
 9:     .org    3
10:     .align  8,$12
11:
12:     .org    3
13:     .align  8,$abcd.w
```

List 2　43　AS.X でのアセンブル例

```
 1: 00000000                          .text
 2: 00000000 =00000003                .org    3
 3: 00000003 0000000000               .align  8
 4: 00000008
 5: 00000008                          .data
 6: 00000000 =00000003                .org    3
 7: 00000003 0000000000               .align  8
 8: 00000008
 9: 00000008 =00000003                .org    3
10: 00000003 1212121212               .align  8,$12
11: 00000008
12: 00000008 =00000003                .org    3
13: 00000003 ABCDABCDAB               .align  8,$abcd.w
```

List 2 44 X680x0 HAS でのアセンブル例

```
 1:  00000000                          .text
 2:  00000000 =00000003                .org    3
 3:  00000003 004E714E71               .align  8    * .text では $4E71
 4:  00000008
 5:  00000008                          .data
 6:  00000000 =00000003                .org    3
 7:  00000003 0000000000               .align  8    * .text 以外では $0000
 8:  00000008
 9:  00000008 =00000003                .org    3
10:  00000003 0000120012               .align  8,$12
11:  00000008
12:  00000008 =00000003                .org    3
13:  00000003 00ABCDABCD               .align  8,$abcd.w
```

Chapter 3

X680x0 HLK

リンカは CPU の変更に影響される部分が少ないため，変更点もバグフィックス等，それほど多くはありません。本章では，X680x0 HLK のバージョンアップの内容，新しいオプションスイッチの紹介をします。

3-1 X680x0 HLK の拡張概要

X680x0 HLK は CPU の依存度が低いため，大幅な変更はありません。本セクションでは，拡張された機能の概要を説明します。

前バージョン同様に 68000 のコードしか使用していませんので，すべての **X680x0** で動作します。

◆HUPAIR 規定に対応

HUPAIR 規定[1]に対応することにより，256 文字以上のコマンドラインを扱うことが可能になりました。もちろん，従来のインダイレクトファイルによるコマンドラインの受け渡しもできます。

1) **HUPAIR** 規定とは，***Human68k User Program Argument Interface Regurations*** の略称で，**Human68k** 上のユーザプログラム間でのコマンドラインの引き渡し，受け取りの方法を規定するものです。この規定に対応しているプログラム間では 256 文字以上のコマンドラインを渡すことが可能になります。この規定は，板垣史彦氏が提唱したものです。

◆アラインメント

従来は，**アラインメント**の値が 2（つまり偶数バイト境界）で固定でした。**X680x0 HLK** では，この値をオプションスイッチで変更することができるようにしました。また，オブジェクトファイル中にアラインメント情報がある場合は，それを利用します。アラインメントにより空いた所には 0 が埋め込まれます。

◆マップファイルへの項目追加

アラインメント値を設定できるようになったので，アラインメント情報の項目が増えました。

◆バグフィックス

以下のバグが修正してあります。

・-o スイッチの動作が正しくなかったのを修正

・rldata セクションのデータを含むオブジェクトファイルをリンクした後は，rdata セクションのデータが正しい位置に書き込まれないのを修正

3-2 拡張された機能

3-2-1 HUPAIR 規定に対応

従来は **Human68k** の仕様により，コマンドラインは 255 文字までしか渡すことができませんでした。そのため，大量のオブジェクトファイルをリンクするために，コマンドラインが 256 文字以上になってしまった場合は，インダイレクトファイルを用意しなければなりませんでした。

しかし，**HUPAIR** 規定に対応することにより，256 文字以上のコマンドラインを扱うことが可能になりました。もちろん，従来のインダイレクトファイルによるコマンドラインの受け渡しもできます。

ただし，COMMAND.X は **HUPAIR** 規定に対応していないため[1]，256 文字以上のコマンドラインを渡すことができません。この機能を生かすためには，**fish.x**[2] や **GNU make** などの **HUPAIR** 規定に対応しているシェルやツールが別に必要となります。

1) 当たり前ですが…

2) **UNIX** 上の csh ライクなシェルで，**HUPAIR** 規定に対応しています。板垣史彦氏作のフリーウェア。

3-2-2 アラインメント

従来は，アラインメントの値が 2[3] で固定でした。**X680x0 HLK** では，この値をオプションスイッチで指定するか，または各オブジェクトファイルのアラインメント情報をもとにして，オブジェクトファイルごとに指定できるようになりました。アラインメントにより空いた所には 0 が埋め込まれます。

デフォルトのアラインメントは，**X680x0 HLK** を実行する環境 (CPU) によって変更されることはなく，常に 2 に固定されています。そのため，**X68030** などに最適なアライメントを行いたい場合は，**X680x0 HLK** のオプションスイッチの指定やアセンブラソース中での指定により明示的に指定する必要があります。

3) つまり偶数バイト境界です。

アラインメントのポリシーは，以下のようになっています。

- **アラインメントの値はオプションスイッチで指定されたデフォルト値よりも，オブジェクトファイル中のアラインメントの値が優先される**

- **アラインメントはオブジェクトファイルのすべてのセクションに対して行われる。たとえばオブジェクトファイル "foo.o" のアライメント値が 4 の場合は, "foo.o" の text, data, bss 等すべてのセクションの先頭アドレスは 4 の倍数になる**
- **各セクションの先頭のアラインメントは, リンクされたオブジェクトファイル中で最大のアラインメント値を使用して行われる**

注　意

アラインメント情報が 2 以外のオブジェクトファイルをリンクした場合, X680x0 HLK 以降とそれ以前では, 生成される実行ファイルの大きさが異なることに気をつけてください。X680x0 HLK 以降では, ファイルサイズが追加されたパディングの分だけ大きくなります。

3-2-3 マップファイルへの項目追加

アラインメントの指定を行えるようになったため, 各オブジェクトファイルごとにアラインメント値 (16 進数) を表示するように追加されました。項目名は "align" (23 行目) です。

マップファイルは以下のようになります。

List 3　1　X680x0 HLK のマップファイル

```
 1:  ==============================================================
 2:  J:¥home¥salt¥develop¥hlk¥main.x
 3:  ==============================================================
 4:  exec              : 00000000
 5:  text              : 00000000 - 00006bc1 (00006bc2)
 6:  data              : 00006bc2 - 00006cf7 (00000136)
 7:  bss               : 00006cf8 - 0000b131 (0000443a)
 8:  common            :
 9:  stack             :
10:  rdata             :
11:  rbss              :
12:  rcommon           :
13:  rstack            :
14:  rldata            :
15:  rlbss             :
16:  rlcommon          :
17:  rlstack           :
18:
19:
```

```
20: ============================================================
21: main.o
22: ============================================================
23: align              : 00000002
24: text               : 00000000 - 000013bd (000013be)
25: data               : 00006bc2 - 00006cf7 (00000136)
26: bss                : 00006cf8 - 00007127 (00000430)
27: stack              :
28: -------------------------- xref -------------------------
29: strcpy             : in string.o
30: strcat             : in string.o
31: strlen             : in string.o
32:
33:                    :
34:                    :
35:                    :
36:                    :
..........................................................................
```

3-3 拡張されたオプションスイッチ

X680x0 HLK では，拡張された機能をサポートするためのオプションスイッチが追加されています。本セクションではその説明をします。なお，今回のバージョンアップにともなって仕様が変更されたり，廃止されたオプションスイッチはありません。

● `-e` *nnnn*　　アラインメント値の設定

-e *nnnn*　デフォルトのアラインメント値を設定する

書式：　`-e` *nnnn*

機能：　デフォルトのアラインメント値を設定します。

解説：　アラインメント値の指定を行っていないオブジェクトファイルに対するデフォルトのアラインメント値を設定します。

nnnn は 2 ～ 256 の数で，2 の N 乗でなければいけません。10 進数で指定します。指定されたアラインメント値が正しくない場合は，“Bad option” エラーが発生します。

例：　アラインメント値を 16 に設定します。

```
A> hlk -e 16 foo.o bar.o
```

Chapter 4

X680x0 GDB

X680x0 GDB は，X68000 シリーズおよび X68030 に対応したソースレベルデバッガです。本章では，X68000 版 GDB を X68030 に対応させた X680x0 GDB について解説します。

4-1 X680x0 GDBの拡張概要

前著「*X68000 Develop.*」では，**GCC**，**HAS**，**HLK**で開発されたプログラムを**GDB**でデバッグするためのコマンドについて解説しました。ここでは，**X680x0 GDB**で追加された機能と変更された機能についてのみ解説することにします。

X680x0 GDBは，**X68000**シリーズおよび**X68030**に対応したソースレベルデバッガです[1)]。**X68030**では，CPUが68000から68030に変更になったため，割り込みやスーパーバイザモードでの動作がだいぶ違ったものになっています。そのため**X680x0 GDB**は起動時に機種を判別し，68000，68030の両タイプのCPUで動作するようそれぞれの動作モードがあります。また，デバッグするプログラムコードについても，68030命令および数値演算コプロセッサ68881，68882の命令にも対応しています。

1) 「*X68000 Develop.*」に付属している**GDB**は，**X68000**を対象に作成してあったため，**X68030**では使用できません。

◆バグフィックス

X68000版**GDB**には，次のようなバグがありました。**X680x0 GDB**では正しく動作するようになっています。

- 式に識別子(シンボル名)と演算子が含まれる場合，それらの間にスペースがなければ“`perse error`”となるバグを修正
- スワップスクリーンモードで半角カナの表示が可能
- スタックフレーム調査の不具合の修正

◆追加した機能

X680x0 GDBには，**X68030**でのCPUの変更にともない新たな次の機能が追加されています。

- **X68000**シリーズおよび**X68030**に対応
- **MC68000**および**MC68030**のプログラムコードのデバッグに対応
- **X68030**で使用する場合に限り，数値演算コプロセッサ**MC68881**，**MC68882**に対応[2)]
- **MC68000**，**MC68030**，**MC68881**，**MC68882**コードの逆アセンブルに対応
- スーパーバイザモードで動作するプログラムのデバッグに対応[3)]
- **LIBC**をリンクしたプログラムのデバッグに対応

2) 「数値演算コプロセッサへの対応」(P.62)を参照してください。この機能は**X68000**では未対応です。

3) 「スーパーバイザモードのデバッグ」(P.63)を参照してください。

◆**変更した機能**

また，**X68000**版**GDB**から変更した機能は次のとおりです。

・**LIBC**の仕様に合わせてシグナルを変更[4]

・チャイルドプロセスの起動方法の変更[5]

4)「シグナルを調査するコマンド」(P.67) を参照してください。

5)「チャイルドプロセスの起動方法」(P.63) を参照してください。

4-2 拡張された機能

X680x0 GDB で追加された機能と変更された機能について解説します。

4-2-1 追加された機能

X680x0 GDB では，X68000 シリーズおよび X68030 に対応するとともに，MC68000 および MC68030 のプログラムコードがデバッグできるようになりました。また数値演算コプロセッサ MC68881，MC68882 や MC68000，MC68030，MC68881，MC68882 コードの逆アセンブルに対応しています。さらに，スーパーバイザモードで動作するプログラムや LIBC をリンクしたプログラムのデバッグに対応させました。

●数値演算コプロセッサへの対応

X680x0 GDB では，数値演算コプロセッサ命令の逆アセンブルが可能になりました。また X68030 で使用する場合に限り，専用ソケットに装着する数値演算コプロセッサ (68881/68882) の内部レジスタの参照[1]と変更が可能です[2]。内部レジスタには CPU レジスタと同様に，X680x0 GDB のコマンドの式からアクセスすることもできます。Table 4-1 に，コプロセッサ内部レジスタと X680x0 GDB でのレジスタ名の対応表を示します。

1) info all-registers コマンドを使用します。詳しくは「FPU レジスタを表示するコマンド」(P.68) を参照してください。

2) 「Human68k ver.2 で使用する場合」(P.71) を参照してください。

Table 4－1● レジスタ名の対応表

コプロセッサ内部レジスタ名	X680x0 GDB レジスタ名
FP0	$fp0
FP1	$fp1
FP2	$fp2
FP3	$fp3
FP4	$fp4
FP5	$fp5
FP6	$fp6
FP7	$fp7
FPCR	$fpcontrol
FPSR	$fpstatus
FPIAR	$fpiaddr

●スーパーバイザモードのデバッグ

X680x0 GDB では，スーパーバイザモードで動作するプログラムをデバッグすることが可能になりました。また，実行中にユーザモードからスーパーバイザモードへ，またはその逆の切り替えも対応可能です。

4-2-2 変更された機能

X680x0 GDB では，チャイルドプロセスの起動方法とシグナルが変更されました。

●チャイルドプロセスの起動方法

従来は内部で処理していたデバッグ対象プログラムの起動と shell コマンドによるチャイルドプロセスの起動方法を，command.x などのコマンドシェルを使用するように変更しました。この変更によって，プログラムに渡されるコマンドラインは指定したコマンドシェルにそのままの形で渡されるため，コマンドラインの解析などはコマンドシェルの機能に依存することになります。つまり X680x0 GDB では，X68000 GDB が内部で行っていたコマンドラインの解析/展開/リダイレクト処理などは，いっさいしないようになっています。このため，HUPAIR 規定に対応したコマンドシェルを使用することで，プログラムには HUPAIR にエンコードしたコマンドラインを渡すことも可能になりました。また，コマンドシェルがリダイレクト処理の機能をもっていれば，コマンドシェルの機能を利用してリダイレクト処理も可能になります。

動作が確認されているコマンドシェルは次のものです。

- **Human68k 付属の** command.x
- **フリーソフトウェアの** fish.x

4-3 廃止されたオプションスイッチ

X680x0 GDB では，以下のオプションスイッチが廃止されています。なお，今回のバージョンアップにともなって仕様が変更されたり，追加されたオプションスイッチはありません。

●`-core`　　コアダンプファイルの指定

X680x0 では意味がないので廃止しました。

●`-exec`　　デバッグ対象プログラムの指定

`.x` 形式の実行ファイルは，シンボリックデバッグ情報を含むことができます。そのため，このオプションスイッチは意味がないので廃止しました。

●`-epoch`　　Emacs を **GDB** のフロントエンドとして使用する
`-fullname`

Human68k がマルチタスクになることを期待して残しておいたのですが，今のところ意味がないので廃止しました。

●`-se`　　デバッグ対象プログラム名の指定

コマンドラインから直接，デバッグ対象プログラムを指定することができるので廃止しました。

●`-symbols`　　シンボルファイルの指定

シンボル情報は実行ファイルに含まれるため，単独で使用することがないので廃止しました。

4-4 変更された環境変数

X680x0 GDB がチャイルドプロセスの起動に使用するコマンドシェルを環境変数 “SHELL”，“SHELL_OPT”，“SHELL_TYPE” に設定します。設定しなかった場合は command.x が使用されます。この場合 command.x のディレクトリは，環境変数 PATH に設定されていなければなりません。

◆環境変数 SHELL

環境変数 SHELL には，使用するコマンドシェルのファイル名を設定します。もし，そのファイルが存在するディレクトリにパスが通っていなければ，ドライブ名およびディレクトリ名も含むフルパスで設定しなければなりません。

◆環境変数 SHELL_OPT

環境変数 SHELL_OPT には，コマンドシェルのコマンドライン引数を設定します。たとえば command.x では “/C” を，**UNIX** から移植したコマンドシェルでは “-C” を指定します。

◆環境変数 SHELL_TYPE

環境変数 SHELL_TYPE には，コマンドシェルのタイプを設定します。コマンドシェルのタイプとは，**X680x0** には command.x 以外にも **UNIX** から移植したコマンドシェルなどがあるため，それぞれコマンドライン引数の渡し方が異なっているからです。**X680x0 GDB** では，command.x タイプと **UNIX** タイプのコマンドシェルが使用できるように，この環境変数で区別しています。

環境変数 SHELL_TYPE の値が，“COMMAND” であれば COMMAND.X タイプ，“**UNIX**” あるいはそれ以外ならば **UNIX** タイプとなります。

4-5 拡張されたコマンド

本セクションでは，拡張されたコマンドについて解説します。

4-5-1 廃止されたコマンド

X680x0 GDB で廃止されたコマンドには次のものがあります。

- **printsyms**　デバッグ情報の出力

  X68000 GDB の printsyms コマンド[1]が廃止され，同様の機能をもつ "maintenance print symbols"[2] コマンドが追加されました。

1) 「*X68000 Develop.*」Vol.1 P.219，Vol.2 P.200 を参照してください。

2) 「メンテナンスコマンド」(P.70) を参照してください。

4-5-2 変更されたコマンド

- info registers　レジスタの調査
- info signals　シグナルの調査

● レジスタの表示

info registers コマンド[3]はレジスタを表示するコマンドですが，次のように表示形式が変更になりました。小文字のレジスタ名はレジスタ変数として式で利用可能ですが，大文字のものはレジスタ変数としては利用できません。書式は以前と同じです。

3) 「*X68000 Develop.*」Vol.1 P.221，Vol.2 P.203 を参照してください。

次に info registers コマンドの実行例を示します。

```
(gdb) info registers
pc:002a41e2 usp:002be8fc ssp:001efc42 sr:0000
S:0 M:0 IML:0 X:0 N:0 Z:0 V:0 C:0
d 00000015 00000004 00000000 00001000
  0029f98b 00000000 00000000 00000000
a 002c0934 002a6be2 0000000a 002a2db4
  002a41f0 0029a120 002be8fc 002be8fc
(gdb)
```

● シグナルを調査するコマンド

LIBC の仕様に合わせて，シグナル名およびシグナル番号を変更しました。LIBC のシグナルにはソフトウェア例外によるものとハードウェア例外によるものがありますが，X680x0 GDB では，ハードウェア例外によるシグナルのみに対応しています。ただし，SIGALRM については LIBC の内部で特殊な扱いがなされているために使用できません。したがって，info signals コマンド[4]を実行するとシグナルの一覧が表示されますが，X680x0 GDB が対応しているシグナルは Table 4-2 に示すものだけです[5]。書式は以前と同様です。

4) 「*X68000 Develop.*」Vol.1 P.212，Vol.2 P.203 を参照してください。

5) 詳しくは Chapter 7「シグナル一覧」(P.155) を参照してください。

Table 4－2 ● X680x0 GDB で使用するシグナル一覧

シグナル名	意　味
SIGILL	不当命令の実行
SIGFPE	浮動小数点演算による例外
SIGBUS	アドレスエラー
SIGSEGV	バスエラー
SIGEMT	未定義割り込み
SIGINT	CTRL+C キーによる割り込み

次の画面は info signals コマンドの実行例です。

```
(gdb) info signals
Signal Stop Print Pass to program Description

Signal 0 (0) Yes Yes No Trace/BPT trap
SIGABRT (1) Yes Yes Yes Abort trap
SIGFPE (2) Yes Yes Yes Floating point exception
SIGILL (3) Yes Yes Yes Illegal instruction
SIGINT (4) Yes Yes No Interrupt
SIGSEGV (5) Yes Yes Yes Segmentation fault
SIGTERM (6) Yes Yes Yes Terminated
SIGALRM (7) No No Yes Alarm clock
SIGKILL (8) Yes Yes Yes Killed
SIGBUS (9) Yes Yes Yes Bus error
SIGSTOP (10) Yes Yes Yes Stopped (signal)
SIGEMT (11) Yes Yes Yes EMT Trap
SIGUSR1 (12) Yes Yes Yes User defined signal 1
SIGUSR2 (13) Yes Yes Yes User defined signal 2

Use the "handle" command to change these tables.
(gdb)
```

4-5-3 追加されたコマンド

X680x0 GDBで追加されたコマンドには次のものがあります。

- ●info all-registers　すべてのレジスタの調査
- ●info mpu　MPUタイプの調査
- ●info process　プロセスの調査
- ●maintenance print msymbols　シンボルのアドレスの出力
- ●maintenance print objfiles　内部情報の表示
- ●maintenance print psymbols　部分的なデバッグ情報の出力
- ●maintenance print symbols　デバッグ情報の出力
- ●maintenance print type　シンボルの詳細な表示

● FPUレジスタを表示するコマンド

info all-registersコマンド[6]は，info registersコマンド[7]で表示されるレジスタの一覧にFPUレジスタを加えて表示します。書式は次のとおりです。

6) 省略型は“i al”です。

7) 詳しくは「レジスタの表示」(P.66)を参照してください。

■書式■

■info all-registers
FPUレジスタを表示する

次にinfo all-registersコマンドの実行例を示します。

```
(gdb) info all-registers
pc:002a41e2 usp:002be8fc ssp:001efc42 sr:0000
S:0 M:0 IML:0 X:0 N:0 Z:0 V:0 C:0
d 00000015 00000004 00000000 00001000
  0029f98b 00000000 00000000 00000000
a 002c0934 002a6be2 0000000a 002a2db4
  002a41f0 0029a120 002be8fc 002be8fc

fpcontrol:00000000 fpstatus:00000000 fpiaddr:00000000
fp0:0 (raw 0x000000000000000000000000)
fp1:0 (raw 0x000000000000000000000000)
fp2:0 (raw 0x000000000000000000000000)
fp3:0 (raw 0x000000000000000000000000)
fp4:0 (raw 0x000000000000000000000000)
fp5:0 (raw 0x000000000000000000000000)
fp6:0 (raw 0x000000000000000000000000)
fp7:0 (raw 0x000000000000000000000000)

(gdb)
```

● MPUタイプを表示するコマンド

info mpuコマンド[8]は，本体に実装されているMPU[9]およびFPU[10]のタイプを表示します。FPUタイプは，MC68881またはMC68882が実装されていなければ表示されません。また X68000では，FPUの扱いが異なるために実装されていても表示されません。書式は次のとおりです。

8) 省略型は "i m" です。

9) *Micro Processer Unit* の略です。

10) *Floating Point Unit* の略です。

—— ■書式■ ——

■ info mpu
MPUタイプを表示する

次の画面は，X68030で動作している場合に info mpu コマンドを実行したものです。

```
(gdb) info mpu
Micro Processer Unit  : MC68030
Floating Point Unit   : MC68882
(gdb)
```

●プロセス情報を表示するコマンド

info process コマンド[11]は，デバッグ対象としてメモリ上にロードされたプログラムについての情報を表示します。この情報から，プログラムのプロセス管理テーブルおよびそのプログラムに割り当てられたプロセスヒープを知ることができます。書式は次のとおりです。

11) 省略型は "i proc" です。

—— ■書式■ ——

■ info process
プロセス情報を表示する

次に info process コマンドの実行例を示します。

```
(gdb) info process
[debugger]
process name: "gdb.x"
psp address : 0x16aad0
heap start  : 0x16abc0
heap end    : 0x298fff

[user]
process name: "test.x" * デバッグ対象プログラム名
psp address : 0x2a40e0 * プロセス管理テーブルのアドレス
heap start  : 0x2a41d0 * プログラムに割り当てられたメモリ
heap end    : 0x2ce90f

(gdb)
```

12)省略型は “mai” です。

● メンテナンスコマンド

maintenance コマンド[12]は，プログラムの制御に使用するためにデバッグ対象プログラムから得られたシンボリックデバッグ情報などにアクセスするためのコマンドです。このコマンドは，show コマンドや info コマンドと同様に，1 つのサブコマンドと引数とで指定します。次にこれらのコマンドを示します。

■書式■

13)省略型は “mai p m” です。

■**maintenance print msymbols <ファイル名>**[13]

X680x0 GDB が読み込んだシンボリックデバッグ情報を簡単な形式で <ファイル名> に出力する

14)省略型は “mai p o” です。

■**maintenance print objfiles**[14]

X680x0 GDB が読み込んだシンボリックデバッグ情報やプログラムに関する情報を格納したアドレスを表示する

15)X680x0 GDB では機能しません。

■**maintenance print psymbols <ファイル名>**

部分的なシンボリックデバッグ情報を出力する[15]

16)省略型は “mai p s” です。

17)「*X68000 Develop.*」Vol.1 P.219, Vol.2 P.200 を参照してください。

■**maintenance print symbols <ファイル名>**[16]

シンボリックデバッグ情報を <ファイル名> に出力する。X68000 GDB の printsyms コマンド[17]と同じ機能をもつコマンド

18)省略型は “mai p t” です。

■**maintenance print type <シンボル名>**[18]

指定したシンボルに関する情報を詳細に表示する

4-6 X680x0 GDB の制限

X680x0 GDB には，従来の制限[1]に加えて次のような制限があります。

1) 「*X68000 Develop.*」Vol.I P.235 を参照してください。

4-6-1 X68030 のレジスタ

X680x0 GDB は **X68000** シリーズおよび **X68030** の両機種に対応したため，**X68030** のレジスタで一部アクセス不可能なものがあります。そのレジスタ名を Tabel 4-3 に示します。

Table 4 – 3 ● X680x0 GDB でアクセス不可能なレジスタ

レジスタ名	レジスタの種類
ISP	割り込みスタックポインタ
VBR	ベクタベースレジスタ
SFC	ソースファンクションコードレジスタ
DFC	デスティネーションファンクションコードレジスタ
CACR	キャッシュコントロールレジスタ
CAAR	キャッシュアドレスレジスタ

MSP(マスタスタックポインタ)は，SSP(スーパーバイザスタックポインタ)として扱っています。

4-6-2 Human68k ver.2 で使用する場合

X68000 では **Human68k** によって DOS コールに割り当てられていた F 系列未実装命令が，68030 ではコプロセッサ命令とされています。このため **Human68k** ver.3.0 では，ぶつかりあいの起こる DOS コールの FF50～FF7F が FF80～FFAF に移動されています。そのため，ver.3.0 以前の **Human68k** では，この部分についての正しい逆アセンブルができません。

4-7 X680x0 GDB が使う割り込み

X680x0 GDB はデバッグ対象プログラムの実行を制御するために，いくつかの割り込み処理をフックしています。**X680x0 GDB** でフックしている割り込みは Table 4-4 のとおりです[1]。

1) 「*X68000 Develop.*」Vol.1 P.212 もあわせて参照してください。

Table 4－4 ● X680x0 GDB で使用している割り込み

ベクタ番号	意　味	用　途
2	バスエラー	デバッグ対象プログラム実行中にバスエラーが発生した場合，デバッガに制御を戻すために使用
3	アドレスエラー	デバッグ対象プログラム実行中にアドレスエラーが発生した場合，デバッガに制御を戻すために使用
4	不当命令	デバッグ対象プログラム実行中に不当な命令を実行した場合，デバッガに制御を戻すために使用
5	0除算	デバッグ対象プログラム実行中に0による除算を実行した場合，デバッガに制御を戻すために使用
6	CHK/CHK2 命令の実行	デバッグ対象プログラム実行中に CHK，CHK2 命令を実行した場合，デバッガに制御を戻すために使用
7	TRAPV 命令の実行	デバッグ対象プログラム実行中に TRAPV 命令を実行した場合，デバッガに制御を戻すために使用
9	トレース	デバッグ対象プログラムをステップ実行するために使用
13	コプロセッサプロトコル違反	デバッグ対象プログラム実行中にコプロセッサプロトコル違反が発生した場合，デバッガに制御を戻すために使用
14	フォーマットエラー	デバッグ対象プログラム実行中にフォーマットエラーが発生した場合，デバッガに制御を戻すために使用
15	未初期化割り込み	デバッグ対象プログラム実行中に未初期化割り込みが発生した場合，デバッガに制御を戻すために使用
31	レベル7割り込みオートベクタ	デバッグ対象プログラム実行中に **X680x0** 本体の INTERRUPT スイッチが押された場合，デバッガに制御を戻すために使用
41	TRAP 9	ブレークポイントを処理するために使用
45	TRAP 13	デバッグ対象プログラム実行中にキーボードから CTRL+C が押された場合，デバッガに制御を戻すために使用

これらの割り込みをデバッグ対象プログラムでフックした場合は，デバッガに制御が戻らなくなりますので注意してください。

また，この他にもチャイルドプロセスの起動方法の変更にともなって，DOSコールのEXECをチャイルドプロセスが起動するまでの間だけフックしています。もし，この間にバックグラウンドプロセスなどでDOSコールのEXECが実行された場合，動作に支障をきたすおそれがあります。

Chapter 5

Develop. 便利帳

本章では，開発ツールを使用するために必要な環境および起動方法について簡単に解説します。

5-1 GCCの概要

GCC は FSF で開発された **UNIX** 上の **C** コンパイラです。それを **X68000** および **X68030** に移植したものが **X680x0 GCC** です。**X680x0 GCC** とは，CPU **680x0** において，おそらく世界最高速のコードを出力するであろう **C** コンパイラです。

5-1-1 GCCが扱うファイル

GCC コンパイラドライバは，次のファイルを扱うことができます。なおこれらは，**X68000 GCC** から変更されていません。

◆Cソースファイル

C プログラムのソースファイルです。拡張子は“.c”です。

◆アセンブラソースファイル

アセンブラのソースファイルです。このソースファイル中に未解決なシンボルがあると，それは外部参照として処理されます。通常，拡張子は“.s”です。**HAS** 拡張機能を用いる場合は，拡張子が“.has”になります。

◆オブジェクトファイル

アセンブラソースをアセンブルすることによって得られるファイルです。このファイルはオプションで特に指定がなければ，clib と gnulib の 2 つのライブラリファイルとともにリンクされます(デフォルト)。拡張子は，“.o”です。

またこの **GCC** コンパイラドライバには，前田氏作成の cshwild ライブラリがリンクされています。このライブラリによって，255 文字を超えるコマンドラインを，**MS-DOS** ふうにいえば応答ファイルの形で渡すことができます。command.x で渡すことができない 255 文字を超えるオプションスイッチは，そのままそのオプションスイッチをファイルにして，次のような書式で渡すことができます。

gcc -+-+-<ファイル名>

コンパイラドライバは指定されたオプションスイッチのほかに，多数の引数をコンパイラに指定することがあります。この場合には，コンパイラドライバが応答ファイルを自動的に作成するので，ユーザが特に意識する必要はありません。しかし，この応答ファイルは環境変数 temp が示すディレクトリに作成されるので，ディスクの空き容量には十分注意してください。

5-1-2 GCCが使う環境変数

GCC はコンパイルを行うときにいくつかの環境変数を参照します。環境変数にはコンパイラが稼動するのに必須のものと，そうでないものがあります。必須の環境変数は，すべて XC が必要とするものと同一です。付属ディスクのインストーラを使用した場合には，これらは自動的に適切に設定されます。オートインストールされなかったユーザは，本セクションを参考にご自分の望まれる環境を構築してください。なおこれらの環境変数は，X68000 GCC から変更されていません。

● 必須の環境変数

以下の環境変数は必ず設定しておいてください。XC が動いている環境ならば，必ず設定されている環境変数です。

◆ PATH, path

コンパイラドライバ gcc.x が，プリプロセッサ gcc_cc1.x，コンパイラ本体 gcc_cc1.x を起動する場合に検索するディレクトリです。ドライバ自体も，この環境変数に指定されたディレクトリに存在しているのが普通です。

◆ include

システムヘッダ[1]がおかれたディレクトリをフルパスで設定します。

1) stdio.h などの標準ヘッダです。

◆ temp

コンパイラが作業用テンポラリファイルを作成するディレクトリをフルパスで指定します。できるだけ高速なディスクドライブを指定します。普通は RAM ディスクドライブを設定しておきます。

● 必須でない環境変数

ここで説明する環境変数は X680x0 独自のものです。環境変数名には移植者の“悪趣味”も反映されています[2]。

2) 笑って許してください。

3) Human68k ではコマンドライン文字数は，普通 255 文字に制限されています。

4) オプションスイッチの詳細については，「*X68000 Develop.*」Vol. 2（P.2～P.47）を参照してください。

5) 512K バイトあります。

6) 256K バイトあります。

◆GCC_OPTION

環境変数“GCC_OPTION”によって，あらかじめいくつかのオプションスイッチを設定しておくことができます。いつも同じオプションスイッチを設定する場合やタイピング節約およびコマンドラインの文字数節約[3]に有効です。

```
set GCC_OPTION= AEFGILMPST
```

GCC_OPTION, AEFGILMPST はすべて大文字で指定しなければならず，“=”と“GCC_OPTION”との間にスペースを入れると無効になります。

また -f がついたオプションスイッチは，それぞれコマンドライン上で，

```
-fno-???
```

と指定すれば，環境変数上での設定を無効にすることができます。その他のオプションスイッチは否定できません[4]。

- ●A　-fforce-addr の指定
- ●E　標準エディタ ed.x が扱える形式のエラーメッセージの生成
- ●F　-fomit-frame-pointer の指定
- ●G　コンパイラが作業用メモリをメインメモリ上で使いつくすとスーパーバイザモードに移行して，GRAM[5]を作業用メモリとして使用する。GRAM の内容は無条件に破壊するので注意が必要
- ●I　-finline-functions の指定
- ●L　-fstrength-reduce の指定
- ●M　-fforce-mem の指定
- ●O　**X68000** 専用の最適化パスを許可する。最終的なコードのループ内部不変定数の移動を行う
- ●P　このオプションスイッチだけほかとは異なり，pea.l 0.w と clr.l -(sp) のコードの選択をする。前者が 2 バイト多くて，2 クロック高速なコードで，デフォルトは pea.l 0.w
- ●S　-fstack-check の指定
- ●T　コンパイラが GRAM を使いつくしたときに，マウスプレーンテキスト RAM を作業用メモリ[6]として利用する
- ●W　-Wall の指定

◆DOSEQU

GCC が生成するアセンブラファイルは，デフォルトでは doscall.equ という **Human68k** のシステムコールを扱うシンボルファイルを使用します。デフォルト以外のファイルを使いたい場合に，ここにそのファイルネームを指定しておけば，コンパイラはそれをそのままアセンブラソースに出力します。このファイルは DOSCALL 関数拡張のために使われます。

◆ SXEQU

環境変数 DOSEQU と同様に，**SX-Window** 開発モードで使われる SXCALL 関数のためのシンボルファイルとして，デフォルトの sxcall.equ 以外のファイルを使う際に，そのファイルネームを指定します。

◆ **真里子，**MARIKO

GCC の拡張[7]を許可するための環境変数です。この設定がない場合の **GCC** は，ほぼ 100% 普通の 68000 CPU のための **GCC** としてふるまいます[8]。

SET **真里子**= ABCDEF

- ●A　2 進ビット表現の拡張，日本語識別子拡張，割り込み関数の記述
- ●B　asm("frame *reg*") の拡張，asm("extern *reg*") の拡張
- ●C　**ソースコードデバッガ**対応
- ●D，E　疑似統合環境の実現
- ●F　コンパイル過程の表示

◆ **満里奈，**MARINA

疑似統合環境で使われるエディタを指定します。環境変数 PATH か path の指定するディレクトリにエディタの実行形式が存在すれば，フルパスで指定しなくてもファイルネームだけで起動することができます。

5-1-3　起動方法と書式

GCC は，コマンドラインから次のような書式で起動します。なお起動方法と書式は，**X68000 GCC** から変更されていません。

gcc **<ファイル名> [<オプションスイッチ>]**

<ファイル名> には “.c”，“.s”，“.has”，“.o” の拡張子をもつファイルが複数個指定できます。コンパイラドライバ gcc.x は各拡張子に応じて必要な処理を順次実行し，最終的に 1 つの実行ファイルを生成します。**<オプションスイッチ>** で特に指定がなければ，一番最初に指定されたベースファイル名に “.x” を付加した名前の実行ファイルを生成します。

<ファイル名> も **<オプションスイッチ>** も指定されなかった場合には，書式を説明するヘルプメッセージが表示されます[9]。また **<オプションスイッチ>** は “-” で始まる文字列で，その順番は **<ファイル名>** も含めて任意に指定することができます。

7) 別の言葉では極悪改造ともいいます。

8) 詳しい拡張内容については，「*X68000 Develop.*」Vol.1 P.48 を参照してください。

9) ヘルプメッセージは，gcc.x が収められているディレクトリに gcc.hlp というファイルがなければ，画面には表示されません。

5-1-4 オプションスイッチ

X680x0 GCC には，次のようなオプションスイッチがあります。これらのオプションスイッチはコマンドラインから直接指定することも，先ほど説明した環境変数に設定しておくこともできます。なお，これらのオプションスイッチのいくつかは **X68000 GCC** から変更/追加/廃止されています。

● `-a`	ブロック単位でのプロファイラ
● `-ansi`	ANSI 違反の報告
● `-C`	コメントを削除しない
● `-c`	オブジェクトファイルの生成
● `-D`	マクロの定義
● `-E`	プリプロセッサ処理結果の出力
● `-f`	最適化の許可／禁止
● `-g`	ソースコードデバッグの生成
● `-I`	インクルードパスの指定
● `-l`	ライブラリの指定
● `-M`	ファイル依存関係の出力
● `-MM`	ファイル依存関係の出力
● `-mregparm`	引数をレジスタ渡しにする
● `-mshort`	`int` を 16 ビットにする
● `-m68881`	68881 用コードの生成
● `-m68020`	68020 ／ 68030 用コードの生成
● `-m68040`	68040 用コードの生成
● `-O`	最適化の実行
● `-o`	出力ファイルネームの指定
● `-p`	プロファイラコードの生成
● `-pedantic`	ANSI に厳密に適合
● `-Q`	バーボーズモード指定
● `-S`	アセンブラソースの生成
● `-traditional`	伝統的な C 言語仕様に準拠
● `-trigraph`	`trigraph` シーケンスの認識
● `-U`	マクロの削除
● `-v`	コマンドラインの表示
● `-version`	コンパイラのバージョン表示
● `-W`	ワーニングの許可／指定されたワーニングの許可
● `-w`	ワーニングの禁止
● `-cc1-stack`	コンパイラスタック量の指定
● `-cpp-stack`	プリプロセッサスタック量の指定

- **-fall-bsr** PC 間接形式の使用
- **-fall-remote** 記憶クラスの固定化
- **-fall-text** テキストセクションですべてを出力する
- **-fansi-only** ANSI で規定された予約語のみを認識する
- **-ffppp** FPPP.X 用コードの生成
- **-ffpu-hard-bug** ハード障害を回避するコードの生成
- **-fignor-cpu-type** CPU チェックコードを挿入せずにコンパイルする
- **-flong-offset** PC 間接命令をすべてロングオフセットにする
- **-fms-dos** 8086 系 C プログラムをコンパイルする
- **-fno-const-mult-expand** 定数乗法展開の禁止
- **-fpic** 変数をすべて A5 ベースの間接アドレッシングにする
- **-frtl-debug** rtl の書き出し
- **-fscd** ソースコードデバッグの生成
- **-fstrings-align** 文字列の偶数整合
- **-fstack-check** スタックチェックコードの生成
- **-fstruct-strict-align** 構造体のパッキング
- **-ftext-report** 詳細なエラー報告
- **-fundump** undump コンパイルの指定
- **-SX** SX-Window プログラムモードの指定
- **-z-heap** ヒープサイズの指定
- **-z-stack** スタックサイズの指定

5-2 HASの概要

アセンブラは，アセンブリ言語で書かれたソースファイルをアセンブルしてオブジェクトファイルを生成します。このオブジェクトファイルをリンカによってリンクすることで，実行可能なファイルを生成することができます。

HASはこのアセンブラの1つで**SHARP**純正コンパイラ**XC**に付属する純正アセンブラAS.Xと上位コンパチブルの機能をもっています。純正AS.Xに比べてアセンブル作業をより高速に行うことができ，さらにいくつかの拡張機能が追加されています。

5-2-1 HASが使うファイル

HASは，アセンブルにあたって以下のファイルを使用します。この中には，アセンブラが参照するファイルやアセンブラ自身が作成するファイルも含まれます。なおこれらは，**X68000 HAS**から変更されていません。

● 入力ファイル

次の2つのファイルはユーザが作成し，アセンブラに対する入力となるファイルです。

◆ソースファイル

アセンブリ言語で書かれたファイルです。拡張子は通常“.s”を使用しますが，**HAS**で拡張されたオリジナルの機能を使用したソースファイルなどは，純正アセンブラAS.Xでアセンブルできないことを明示する意味で，拡張子に“.has”を使用することができます。

◆インクルードファイル[1]

これもソースファイルの一種ですが，そのなかで他のソースファイル中から .include 疑似命令によって，参照されるファイルのことをいいます。インクルードファイルはソースファイルと同じディレクトリにおくこともできますが，他のディレクトリにまとめておいておき，環境変数 include でそのディレクトリを指定するか，またはアセンブルの際のコマンドライン上の -i スイッチによってディレクトリを指定することもできます。

1) インクルードファイルの拡張子は特に定められていませんが，“.h”はC言語プログラムのヘッダファイルと混同する恐れがあるため，一般的には“.mac”，“.equ”，“.inc”などが使われているようです。

● 出力ファイル

次の4つのファイルは，入力されたファイルをもとにアセンブラ自身が作成するものです。

◆オブジェクトファイル

ソースファイルをアセンブルすることによって得られるファイルです。ファイル名は，特に -o スイッチで指定しないかぎり，ソースファイル名の拡張子を“.o”に変更したファイル名となります。

◆リストファイル

アセンブルリストを出力するファイルで，-p スイッチを指定することで作成されます。-p スイッチでファイル名を指定しないかぎり，リストファイルはソースファイル名の拡張子を .prn に変更したファイル名で作成されます。

◆シンボルファイル

ソースファイルのなかで使用されたシンボル名に関する情報を並べたファイルで，-x スイッチを指定することで作成されます。-x スイッチでファイル名を指定しないと，シンボルファイルの内容は画面上に出力されます。

◆テンポラリファイル

アセンブラ自身が使用する作業用ファイルです。通常，作業はすべてメモリ上で行われますが，メモリ容量が不足した場合に，一時的にこのファイルを作成することがあります。アセンブルが終了すると，テンポラリファイルは自動的に削除されます。

通常，テンポラリファイルはカレントディレクトリ上に作成されますが，環境変数 temp が設定されているときはそのパス上に，またアセンブラのコマンドラインで -t スイッチが指定されると，このスイッチによって指定されたパス上に作成されます。

5-2-2 HAS が使う環境変数

HAS が参照する環境変数には，以下のものがあります。なおこれらの環境変数は，**X68000 HAS** から変更されていません。

◆include

標準的なインクルードファイル[2)]がおかれているディレクトリをフルパスで指定します。

2) 具体的には，IOCS コールや DOS コールの定義ファイルである，IOCSCALL.MAC，DOSCALL.MAC などです。

◆**temp**

アセンブラがテンポラリファイルを作成するディレクトリをフルパスで指定します。

3) "HAS" は大文字のみで，小文字は受けつけません。

◆HAS[3)]

HAS が常に使用するオプションスイッチを設定します。設定されたオプションスイッチは，**HAS** を実行するときにコマンドラインの最後に追加されます。

5-2-3 起動方法と書式

HAS は，コマンドラインから次のような書式で起動します。なお起動方法と書式は，**X68000 HAS** から変更されていません。

HAS [**<オプションスイッチ>**] **<ファイル名>**

<オプションスイッチ> には，アセンブラの動作を制御したり，必要な情報を付加するオプションスイッチを指定します。オプションスイッチの指定は "-" から始めますが，"/"(スラッシュ) から始めることもできます。

<ファイル名> には，アセンブルを行うソースファイル名を指定します。ソースファイル名の拡張子が "**.s**" または "**.has**" である場合は，拡張子を省略することができます。その際，拡張子 "**.s**" と "**.has**" の両方のファイルが存在するときには，拡張子が "**.has**" であるファイルが優先されます。

<オプションスイッチ> の指定がまちがっていたり，**<ファイル名>** が指定されなかった場合には，書式を説明するヘルプメッセージが表示されます。

また，常に使用するオプションスイッチをあらかじめ環境変数 HAS に設定しておくことで，アセンブルの際には必ずそのオプションスイッチが指定されているものとすることができます。このとき，環境変数 HAS の 1 文字目に "*" を設定すると，オプションスイッチ指定キャラクタとしての "/" の使用を禁止することができます。たとえば，コマンドライン上で次のように実行したとします。

```
A>set HAS=-n -m68000
```

上記のように，環境変数 HAS に "-n -m68000" を設定しておくと，最適化禁止，68000 CPU 指定の状態でアセンブルします。

なお **HAS** のもつオプションスイッチの機能は，すべて純正アセンブラ AS.X の上位コンパチブルなので，**HAS** のファイル名 HAS.X を AS.X に変更して使用することで，純正アセンブラとまったく同じように使用できるよ

うになります。

5-2-4 オプションスイッチ

X680x0 HAS には，以下のようなオプションスイッチがあります。これらのオプションスイッチは，コマンドライン上から直接指定することも，環境変数 HAS に設定しておくこともできます。また機能拡張にともなって，オプションスイッチのいくつかは **X68000 HAS** から変更/追加/廃止されています。

- `-8` シンボルの識別長の指定
- `-b` ロングワードの PC 間接を絶対ロングにする
- `-c` **HAS** v2.x 互換の最適化を行う
- `-d` 全シンボルの外部定義指定
- `-e` 外部参照オフセットのデフォルトをロングワードにする
- `-f` リストファイルのフォーマット指定
- `-g` **SCD** 用デバッグ情報の出力
- `-i` インクルードファイルのパス指定
- `-l` タイトル表示の指定
- `-m` アセンブル対象命令セットの指定
- `-n` 最適化の禁止
- `-o` オブジェクトファイル名の指定
- `-p` リストファイルの作成
- `-s` シンボルの定義
- `-t` テンポラリファイルのパス指定
- `-u` 未定義シンボルの外部参照指定
- `-w` ワーニングレベルの指定
- `-x` シンボル情報の出力指定

5-3 HLKの概要

リンカは，アセンブラで作成された**オブジェクトファイル**と**ライブラリファイル** をまとめて１つの**実行ファイル**を作成します。オブジェクトファイルには，**外部参照シンボル**や**外部定義シンボル**の情報が入っています。外部参照シンボルは，それ自身ではシンボルの値を決定できませんので，同じ名前の外部定義シンボルの値が必要になります[1]。

リンカは，これらすべての外部参照シンボルが同じ名前をもつ外部定義シンボルと対応がとれるように，オブジェクトファイルをリンクして実行ファイルを作成します。

1)「*X68000 Develop.*」Vol.I P.88 を参照してください。

5-3-1 HLKが使うファイル

HLKではいくつかのファイルを使用します。ここでは，それらのファイルを入力ファイルと出力ファイルに分けて説明をします。

● 入力ファイル

次の３つのファイルは，HLKで実行ファイルを作成するときに，HLKに与えるファイルです。

◆オブジェクトファイル

アセンブラソースをアセンブラに与えることによって作成されるファイルです。このファイルは，大まかにいって「**外部参照シンボル**」,「**外部定義シンボル**」,「シンボリックデバッグ情報」,「プログラム本体の情報」から構成されています。ファイルの**拡張子** は，“.o”です。

このファイルをリンカに与えることによって，実行ファイルが作成されます。

◆ライブラリファイル

ライブラリファイルは，複数のオブジェクトファイルをまとめたファイルです。よく使用されるオブジェクトファイルをまとめておき，このファイルをリンカに渡せば，リンカは必要なオブジェクトファイルだけをリンクしてくれます。

ライブラリファイルには，**アーカイブ形式**[2]と**ライブラリアン形式**[3]の２種類あります。アーカイブ形式はオブジェクトファイルをまとめて１つの

2) 拡張子は，“.a”です。
3) 拡張子は，“.l”です。

ファイルにしたもので，ライブラリアン形式はアーカイブ形式にシンボルのインデックスが付属している形になっています。

ライブラリファイルとオブジェクトファイルの相違点は，ライブラリファイルが与えられたとき，リンカはライブラリファイル中の必要なオブジェクトファイルをリンクするということです。オブジェクトファイルは，指定すれば必ずリンクされます。

極端な話，外部定義シンボルをもたないオブジェクトファイルで構成されているライブラリファイルは，リンカに与えても決してリンクされません。

◆インダイレクトファイル

リンカは，コマンドラインからオブジェクトファイル名やオプション等を読み取る代わりに，インダイレクトファイルからも読み取ることができます。インダイレクトファイルを使用することにより，255 文字を超えるようなコマンドラインが指定できたり，長いコマンドラインを指定する手間を省くことができます。

このファイルは，普通のテキストファイル形式です。スペース，タブ，改行コードはオプションやファイル名の区切り記号になります。

● 出力ファイル

次の 2 つのファイルは **HLK** が出力するファイルです。

◆実行ファイル

実行ファイルは，リンカに与えられたオブジェクトファイル，アーカイブファイルから作成されます。拡張子は，“.x” になります。実行ファイルには，プログラム本体やプログラムをデバッグするための情報[4)]が入っています。

4) シンボル情報やシンボリックデバッグ情報のことです。

◆マップファイル

マップファイルは，オプションを指定することにより実行ファイルとともに作成されるファイルです。このファイルには **HLK** により作成された実行ファイルが，どのオブジェクトファイルから構成されているかや，ラベルの定義，参照情報など実行ファイルに関する情報が書かれています。List 5-1 にマップファイルの例を示して，このファイルの読み方を説明します。

List 5 1 マップファイルの例

```
1:  ===========================================================
2:  prog.x
3:  ===========================================================
4:  exec                       : 0000051c
5:  text                       : 00000000 - 0000087f (00000880)
6:  data                       : 00000880 - 00000995 (00000116)
7:  bss                        : 00000996 - 000019fd (00001068)
```

```
 8:  common                   : 000019fe - 00001acb (000000ce)
 9:  stack                    :
10:  rdata                    : 00000000 - 0000001d (0000001e)
11:  rbss                     :
12:  rcommon                  :
13:  rstack                   :
14:  rldata                   :
15:  rlbss                    :
16:  rlcommon                 :
17:  rlstack                  :
18:
19:
20:  ============================================================
21:  bar.o
22:  ============================================================
23:  align                    : 00000002
24:  text                     : 00000000 - 000006ab (000006ac)
25:  data                     :
26:  bss                      : 00000996 - 00000a5d (000000c8)
27:  stack                    :
28:  rdata                    : 00000000 - 0000001d (0000001e)
29:  -------------------------- xref --------------------------
30:  symbol_from_foo          : in foo.o
31:  symbol_of_common         : in bar.o
32:  -------------------------- xdef --------------------------
33:  symbol_from_bar          : 00000000 (text    )
34:  -------------------------- comm --------------------------
35:  symbol_of_common         : 000019fe - 00001acb (000000ce)
```

マップファイルは大きく分けて，実行ファイルの情報とオブジェクトファイルの情報の 2 つから構成されています。各項目について説明します。カッコ内の行番号は，List 5-1 での位置です。

◆実行ファイルの情報

- **実行ファイル名**　(1 ～ 3 行)
  実行ファイルの名前
- **実行アドレス**　(4 行)
  実行ファイルを起動したときに最初に実行されるアドレス
- **各セクションの位置と大きさ**　(5 ～ 17 行)
  セクション名[5]とそのセクションが占める範囲。カッコ内はセクションのサイズを表す。セクション名だけの行は，そのセクションが使用されていないことを示す

5) 「*X68000 Develop.*」Vol.1 P.87 を参照してください。

◆ **オブジェクトファイルの情報**

- **オブジェクトファイル名** (20 ~ 22 行)
  オブジェクトファイルの名前。ライブラリ中のオブジェクトファイルの場合は，横にライブラリファイルの名前がつく
- **アラインメント値** (23 行)
  オブジェクトファイルの各セクションのアラインメント値
- **各セクションの位置と大きさ** (24 ~ 28 行)
  セクション名とそのセクションが占める範囲。カッコ内はセクションのサイズを表す。セクション名だけの行は，そのセクションが使用されていないことを示している。相対セクションは，使用されているセクションのみ情報が出力される
- **外部参照シンボル** (29 ~ 31 行)
  オブジェクトファイル中で外部参照しているシンボルのリスト。シンボル名と外部参照しているシンボルが定義されているオブジェクトファイル名が出力される。common セクションのシンボルも出力されるが，これは仕様である
- **外部定義シンボル** (32 ~ 33 行)
  オブジェクトファイル中で外部定義しているシンボルのリスト。シンボル名と外部定義しているシンボルの値と属性[6]が出力される
- **コモンエリア** (34 ~ 35 行)
  オブジェクトファイル中で使用しているコモンエリア[7]のシンボルのリスト。シンボル名と，そのシンボルが占める範囲と大きさが出力される。大きさは，必ずしもそのファイルで定義した大きさにはならない。なぜならば，もっと大きい領域を定義しているオブジェクトファイルがあるかもしれないからである

6) 「*X68000 Develop.*」Vol.1 P.155 を参照してください。

7) 「*X68000 Develop.*」Vol.1 P.89 を参照してください。

5-3-2 HLK が使う環境変数

基本的に **HLK** は環境変数を設定しなくても使用することができますが，次に示すような環境変数を設定することによって，より使用しやすくなります。なお，環境変数は大文字／小文字を区別するので注意してください。なおこれらの環境変数は，**X68000 HLK** から変更されていません。

◆ SILK (silk)

この環境変数 SILK または silk の内容は，**HLK** に指定されたコマンドラインの最後につけ加えられます。好みに応じて，自分がよく使うようなオプションスイッチや**コンパイラドライバ**[8]ではリンカに渡すことができないオプションスイッチを指定することができます。

環境変数 SILK が設定されている場合は，環境変数 silk の内容は参照されません。これら 2 つの環境変数が設定されていなくても，**HLK** は正常

8) たとえば，gcc.x や cc.x などです。

に動作します。

◆lib

環境変数 lib は，HLK に -l オプションを指定することで参照されるようになります。環境変数 lib にライブラリのディレクトリ名を設定しておくことで，ライブラリをリンクするときにフルパスで指定しなくても，ライブラリのファイル名だけですむようになるので，コマンドラインを短く，すっきりさせることができます。

環境変数 lib が設定されていない場合[9]は，-l オプションを使用すると，次の図のようなエラーメッセージが出力されます。また，-l オプションが有効でないため clib.a を見つけることもできません。そのため，さらに余分なエラーが発生します。

9) 設定していない人は，あまりいないと思いますが。

```
A>set
A>
A>hlk main.o -l clib.a
Undefined environment variable 'lib'
Not found : clib.a
Undefined symbol(s) in a.o
__main

A>
```

5-3-3 起動方法と書式

HLK はコマンドラインから次のような書式で起動します。なお起動方法と書式は，X68000 HLK から変更されていません。

HLK [<オプションスイッチ>] <ファイル名> [<ファイル名> ...]

<オプションスイッチ> を指定することにより HLK の動作を変更することができます。オプション文字は大文字／小文字の区別をしていません。LK ではオプションスイッチの先頭の文字は，“-”(ハイフン) もしくは “/”(スラッシュ) になっていましたが，HLK ではスラッシュをファイル名の一部とみなすようになっているので，オプションスイッチの先頭の文字は，“-” だけになります。

引数が必要なオプションスイッチの場合[10]，オプションスイッチと引数の間に空白文字を入れなくてもかまいません。ただし，“-ax” のように複数のオプションスイッチをまとめて指定することはできません。この場合は，“-a -x” のように指定します。

10) たとえば，“-i” スイッチなど。

ファイル名には，オブジェクトファイル，ライブラリファイルが指定でき

ます。拡張子を省略した場合，“.o”を拡張子とみなします。またオブジェクトファイル名の先頭が“+”の場合は，“+”を取り除いたオブジェクトファイルを最初にリンクするようにします。オブジェクトファイルが複数指定された場合は，最後に指定されたオブジェクトファイルが先頭になります。

コマンドラインに何も指定しなかったり，オブジェクトファイルを指定しなかった場合は，簡単な説明とタイトルを表示します[11)]。**HLK** のバージョンを確認したい場合は，コマンドラインに何も指定しないで起動すれば確認できます。

11)**HLK** の名前が LK になっているのは，筆者が HLK.X を LK.X にリネームして使用しているためです。

環境変数 SILK または silk が定義されていると，その内容がコマンドラインの最後につけ加えられます。よく使うオプションスイッチやコンパイラドライバでは **HLK** に渡すことができないオプションスイッチ[12)]を，常に指定することができます。

12)たとえば，“-t”スイッチなど。

以下のように設定すると，**HLK** の起動時にタイトルを表示するようになります。

```
A>set SILK=-t
A>lk main.o
X68k SILK Hi-Speed Linker v2.** Copyright 1989-92 SALT
A>
```

5-3-4 オプションスイッチ

HLK で用意されているオプションスイッチを以下に示します。なおこれらのオプションスイッチには，**X68000 HLK** から追加されたものも含まれています。

- ●-a　実行ファイルの拡張子省略時に“.x”をつけない
- ●-d　外部定義シンボルの登録
- ●-e　アラインメント値の設定
- ●-i　インダイレクトファイルの指定
- ●-l　ライブラリパスの使用
- ●-m　最大シンボル数の設定
- ●-o　実行ファイル名の指定
- ●-p　マップファイルの作成
- ●-s　セクション情報を実行ファイルに埋め込む
- ●-t　起動時にタイトルを表示する
- ●-v　バーボーズモードの指定
- ●-w　ワーニングメッセージの抑制
- ●-x　シンボルテーブルの出力禁止
- ●-z　-v オプションを無効にする

5-4 GDBの概要

本セクションではデバッグ対象プログラムをGDBでデバッグする前に，あらかじめ準備しておかなければならない点について説明し，次にHuman68kのコマンドラインからGDBを起動する方法を説明します。そして最後に，GDBを終了する方法について説明します。

5-4-1 プログラムをデバッグできるように準備する

GDBでプログラムをデバッグするには，プログラムをコンパイル／リンクする際に，必要なすべてのデバッグ情報を生成するようにコンパイラに指示する必要があります。

GCCを使った場合は，コンパイラに対して“-g”オプションを指定してコンパイルします。このとき，同時に最適化オプションを指定することもできますが，“-fomit-frame-pointer”オプションは指定しないほうがいいでしょう。このオプションスイッチを指定すると，コンパイラは必要のないスタックフレームを生成しなくなりますので，ローカル変数の参照ができなくなる恐れがあります。また，デバッグしたい関数はインライン展開しないようにしてください。なぜならば，インライン展開した関数はステップ実行できなくなるからです。

5-4-2 GDBが使うファイル

プログラムのデバッグには，次に示すファイルが必要です。なおこれらは，X68000 GDBから変更されていません。

◆ソースファイル

デバッグ対象プログラム[1]のCで書かれたファイルのことです。

1) 以下，プログラムと表記します。

◆実行ファイル

プログラムのソースファイルをコンパイルして作成されたファイルのことです。このファイルには，シンボリックデバッグ情報が含まれていなければなりません。

5-4-3 GDB が使う環境変数

X680x0 GDB は以下に示す環境変数を使用します。ただし，必ずしもこれらを必要とはしません。

◆ HOME

GDB が起動するときに読み込む “.gdbinit” ファイル[2]がおかれているディレクトリをフルパスで指定します。

2) Ext 氏の TwentyOne.x が常駐されていなければ，“_gdbinit” です。「*X68000 Develop.*」Vol.I P.226 を参照してください。

◆ GDB_OPTION

GDB を起動するときに指定するコマンドラインオプションを，そのままこの環境変数で指定することができます。

また，チャイルドプロセスの起動方法の変更により，設定する環境変数が追加されました[3]。**X680x0 GDB** がチャイルドプロセスの起動に使用するコマンドシェルは，環境変数 “SHELL”，“SHELL_OPT”，“SHELL_TYPE” に設定します。設定しなかった場合は command.x が使用されます。この場合，command.x のディレクトリは環境変数 PATH に設定されていなければなりません。

3) Chapter 4「変更された環境変数」(P.65) を参照してください。

◆ SHELL

使用するコマンドシェルのファイル名だけを設定します。もし，そのファイルが存在するディレクトリにパスが通っていなければ，ドライブ名およびディレクトリ名も含むフルパスで設定しなければなりません。

◆ SHELL_OPT

コマンドシェルのコマンドライン引数を設定します。

◆ SHELL_TYPE

コマンドシェルのタイプを設定します。コマンドシェルのタイプとは，**X680x0** には，command.x 以外にも **UNIX** から移植したコマンドシェルなどがあるので，それぞれコマンドライン引数の渡し方が異なっているからです。**X680x0 GDB** では command.x タイプと **UNIX** タイプのコマンドシェルが使用できるように，この環境変数で区別しています。

5-4-4 起動方法と書式

GDB はコマンドラインから次のような書式で起動します。なお起動方法と書式は，**X68000** 版 **GDB** から変更されていません。

gdb [<オプションスイッチ>] <ファイル名>

<オプションスイッチ> には，引数を指定します (省略可能)。そして，<ファイル名> には，プログラムの名前を“.x”拡張子を含めて指定します。また SHARP のデバッガ“db.x”などのように，プログラムに渡すコマンドラインを続けて指定することはできません。

5-4-5 オプションスイッチ

GDB では，起動時に次のオプションスイッチが使用できます。なおオプションスイッチのいくつかは，X68000 版 GDB から廃止されています。

●-batch	バッチ処理
●-cd	ワーキングディレクトリの指定
●-command	コマンドファイルの指定
●-directory	ソースディレクトリの指定
●-help	ヘルプメッセージの表示
●-mem	メモリの設定
●-nx	初期化ファイル.gdbinit を読み込まない
●-quiet	タイトルの非表示
●-remote, -r	リモートコンソールモードで起動する
●-swap	スクリーンスワップモードで起動する
●-tty	標準入出力先の指定

5-4-6 GDBのコマンド入力

ここでは，GDB が起動した後のコマンドの入力について説明します。なおこれらコマンドの入力方法は，X68000 版 GDB から変更されていません。

● コマンド入力

GDB はコマンドの受け入れ準備ができると，プロンプト“(gdb)”[4]を表示します。ユーザはこのプロンプトに続いて，コマンド名とそのコマンドの引数を入力することで， GDB に命令することができます。GDB はコマンドの実行が終了すると，再びプロンプトを表示し，ユーザからのコマンド入力を待ちます。

● コマンド行編集

コマンド入力中は，テキストエディタ感覚で行編集が行えます。つまり，入力中の文字列をカーソルキーを使ってカーソルを移動し，文字を挿入したり削除するといったことができます。またヒストリ機能[5]を使えば，前に実行したコマンドを呼び出して編集／実行することもできます。

4) これはデフォルトの設定です。プロンプトはユーザが自由に変更することができます (Appendix B P.192“set prompt”を参照)。

5) 前に入力したコマンドを使用することができます。「*X68000 Develop.*」Vol. 2, P.211（マニュアル版では P.212）を参照してください。

● コマンド名の省略形

GDBのコマンドはほとんどが長い名前なので，コマンド名をすべて入力しなくても，1文字や少ない文字数の省略形で入力できるようになっています。コマンドはあいまいにならないかぎりで，省略することができます。もしも入力したコマンドの省略形が，複数のコマンドにマッチする場合は，それらのコマンド名をすべて表示します。ただし，あいまいな省略形を特別に許可しているものも存在します。たとえば，省略形の“s”は，他の“s”で始まるコマンドをさしおいて“step”コマンドを意味するように設定してあります。

● コンプリーション機能

GDBでは，コンプリーション機能が使用できます。コンプリーション機能とは文字列を完全に入力しなくても，入力途中で CTRL + I[6]を押せば，GDBのほうで現在参照可能なシンボル名から検索し，ユーザが途中まで入力した文字列を完全な文字列に完成してくれる機能のことで，文字数の多いシンボル名[7]を入力する際にとても便利です。この機能はGDBのコマンド名に対しても有効ですが，コマンド名の入力には，やはり省略形を使ったほうが便利でしょう。

6) コントロールキーを押しながら“I”を押します。

7) 関数，変数名のことです。

● 繰り返し実行

GDBに空行[8]を入力するということは，直前のコマンドをそのまま繰り返すことを意味します。ただし一部のコマンドについては，この方法による繰り返しを許可していません。なぜならば，無意識による繰り返しがトラブルを引き起こしたり，繰り返し実行することが無意味なコマンドなどがあるからです。また，別のあるコマンドでは，繰り返されたときにさらに便利なように，異なった動作をします。たとえば，ステップ実行のコマンド“step”は，一度実行した後は ↵ を押していくだけでステップ実行を進めていくことができるようになっています。

8) プロンプトに対して何も入力しないで，そのままリターンキーを押します。

5-4-7 プログラムの実行

ここでは，プログラムをGDBに対して指定する方法とそのプログラムのデバッグを開始するためにプログラムを実行させる方法について説明します。なおこれらは，X68000版GDBから変更されていません。

● プログラムの指定

デバッグを始めるには，まずGDBにシンボリックデバッグ情報を読み込ませる必要があります。そのためには，デバッグするプログラムのファイルを指定しなければなりません。プログラムを指定する方法には，次の2種類があります。

◆**起動時に引数によるファイル名の指定**

GDB のコマンドラインオプションにファイル名を指定する場合は，次のように入力して GDB を起動します。

gdb <ファイル名>

この場合は，起動時にプログラムのシンボル情報が読み込まれます。

◆**起動後のコマンドによるファイル名の指定**

通常は GDB 起動時にプログラムを指定しますが，プログラムが複数個あった場合など，別のプログラムへと変更することがあります。また，GDB 起動時にデバッグしたいプログラムを記述し忘れたり，ファイル名をまちがえて記述したりすることもあります。これらの場合，いちいち GDB を終了させて起動し直すのはめんどうですから，新しくプログラムを指定します。コマンドによるファイル指定には，次に示す 3 つの方法があります。

・exec-file <ファイル名>
プログラム名を指定します。もしも，ファイルをワーキングディレクトリで見つけることができなかった場合，GDB は環境変数 PATH に従ってコマンド検索対象ディレクトリからファイルを探します。つまり，カレントディレクトリにプログラムが存在しなくてもよいということです。また，ファイル名を絶対／相対パスで指定することもできます。

・symbol-file <ファイル名>
ファイルからシンボルテーブルの情報を読み込みます。このコマンドも "exec-file" コマンドと同じく，ワーキングディレクトリで見つからなければ環境変数 PATH に従って検索します。

・file <ファイル名>
このコマンドは，"exec-file" コマンドと "symbol-file" コマンドを一度に実行させるためのコマンドです。通常，プログラムを指定する場合は，このコマンドを使います。

5-4-8 GDB を終了する

GDB を終了するコマンドは，"quit" です。プロンプトに対して，

quit

と入力することで GDB を終了させることができますが，SX–Window アプリケーションのデバッグ中はこのコマンドを使ってはいけません。必ず，SX–Window アプリケーションを終了し，SX シェルも完全に終了させてから "quit" コマンドで GDB を終了させるようにしてください[9]。

9) GDB は，プロセスを強制終了させるために DOS コールの EXIT を使用しているので，SX–Window アプリケーションを終了させることができません。

X 6 8 k

Programming Series

(#2)

X680x0 libc

はじめに

LIBCはSHARPのパーソナルコンピュータX68000のために，完全にフリーな*ANSI C*ライブラリとして1993年に公開されました。このライブラリはすべてのソースコードが公開されており，自由に使用，配布，改変が許されています。また，同時にペーパーマニュアルとしてソフトバンクより「*X68k Programming Series #2 X680x0 libc*」が刊行されました。

しかしソフトウェアとして宿命的なことに，このライブラリもいくつかのバグがあったため，結果として完全には動作しない部分がありました。しかし，こうしたバグおよび不具合，仕様の不完全さは***Project LIBC Group***の手によって修正されています。これは1993年公開時にはバージョン1.0.20だったものが，現在はバージョン1.1.31であることからもわかると思います(それだけ直す部分が多かったのかと問われれば，「その通りです」としかいえません)。

不具合の多くは，パソコン通信を通して多くのユーザの皆さんから報告されたものです。また，新しい機能の追加や具体的な修正まで送っていただいたことも少なくありません。多くの方々の協力によって，LIBCは成長することができたといえるでしょう。この場をお借りして，協力してくださった皆様にお礼申し上げます。

さて，本書「*X680x0 Develop. & libc II*」は，1993年に公開した最初のLIBCから現在のバージョンにいたるまでの主な変更点や追加機能について述べるとともに，説明が不足していた部分についての補足の意味もかねて出版することになりました。また，最新バージョンである1.1.31のバイナリをパソコン通信以外の方法で配布する目的もあります。私たちとしては，本書がユーザ諸氏のお役に立てることを願っております。

1994年8月

Project LIBC Group

Chapter 6

最新版の概要

LIBC は，1994 年 6 月現在バージョン 1.1.31 が最新版として公開されています。このバージョン 1.1.31 は付属ディスクに収録されていますが，もしさらに最新のものをお探しの場合は，NIFTY-Serve SHARP User's フォーラム・ワークステーション館 (FSHARP3) をお探しになることをお勧めします。「*X68k Programming Series #2 X680x0 libc*」で宣言したとおり，LIBC のサポートは FSHARP3 にて行っています。同様に，LIBC についての質問，要望，不具合報告などは NIFTY-Serve の ID PGA01555 あるいはインターネットアドレス libc-support@sml.co.jp まで電子メールにてお寄せください。できる範囲でサポートいたします。

6-1 修正した機能

バージョン 1.0.20 から 1.1.31 までの間に行ったすべての修正や変更については，ソースファイルに同梱してある“CHANGELOG”ファイル（そのうちで古いものについては“doc/CHANGELOG.10”）にすべて記録してあります。詳細が知りたい場合はそちらを別途参照していただくとして，ここでは特に大きな修正点について解説します。

◆ ファイルオープン[1]

LIBC では同時にオープンできるファイルの制限を <limits.h> に定義した _POSIX_OPEN_MAX で制限していますが，従来はこの制限値が現実的にみて非常に少ない値だったので，しばしばファイルのオープンに失敗することがありました。特に子プロセスを起動している場合などは，ファイルを多くオープンしているので，このエラーが頻発していました。現在の LIBC では，この制限値を十分大きく取ってあります。

1)「*X680x0 libc*」 Vol.2 P.226 を参照してください。

◆ ファイルクローズ[2]

LIBC ではプロセスが終了するときにすべてのファイルをクローズします。しかし，従来は最初からオープンされている標準入出力のファイルなどもクローズしていたため，子プロセスとして起動された場合に親プロセスのファイルまでクローズしてしまうなど，影響をおよぼしていました。現在は，自プロセスでオープンしたファイルだけをクローズするように変更しました。

2)「*X680x0 libc*」 Vol.2 P.40 を参照してください。

◆ テキストモードのサポート

LIBC ではテキストモードをサポートしていますが，従来のバージョンでは十分ではありませんでした。たとえば EOF コード (0x1A) を認識しなかったり，CR，LF コードの自動変換による読み書きサイズの変化などを正しく扱えません[3]でした。現在はそれらの問題を解決し，***MS-C 7.0*** 互換のレベルで，テキストモードでのファイル操作を行えるようになっています。

3) 不具合もあれば仕様上の問題もあります

6-2 変更した機能

本セクションでは変更した機能について解説します。今回の変更によってソースコードの互換性を失う場合もありますが，基本的には「誤った仕様」を訂正するための仕様変更ですから，実質的な問題はないものと思います。また，仕様変更以外にも実装方法の変更などもあります。当然ですが，これらの変更点についてはインタフェイスの変更はありませんので，従来と同じ方法で使うことができます。

6-2-1 ヘッダファイルの変更

次の 2 つのヘッダファイルが変更されています。

- <sys/dos_i.h>
- <sys/iocs_i.h>

● <sys/dos_i.h>

DOS コールライブラリのインライン展開用ヘッダは，従来インラインアセンブラを用いて実装していましたが，**X680x0 GCC** がもっている拡張機能である DOSCALL プロトタイプ宣言を使用するように変更しました。その結果，今までのライブラリよりも，若干コードの質がよくなりました。ただし一部の複雑な関数については，これまでと同様インラインアセンブラで展開されるので，ユーザが使用する場合には，ほんの少し注意が必要です。できればインライン展開用ヘッダは使用せずに，libdos.a を使用するほうが安全でしょう。

● <sys/iocs_i.h>

IOCS コールライブラリのインライン展開用ヘッダは，従来通りインラインアセンブラを用いて実装しています。当初の不具合や問題点の多くは解決しましたが，それでも，ある特定の状況で誤ったコードになることがあります。しかしこの問題は，インラインアセンブラを用いる限り回避できません。そのため，どうしてもインライン展開させたい場合を除けば，libiocs.a を使用するほうが安全です。

6-2-2 関数仕様の変更

以下の関数が変更されています。

- access 関数
- fread 関数
- free 関数
- fwrite 関数
- getcwd 関数
- getdcwd 関数
- main 関数
- read 関数
- readdir 関数
- stat 関数
- lstat 関数
- fstat 関数
- write 関数

● access 関数

stat 関数が，stat 構造体[1]の st_mode メンバを生成する条件を変更したのにともない，access 関数[2]でも拡張子“.Z”は実行属性とみなさないように仕様変更しました。現実的に，“.Z”という拡張子の実行ファイルを扱うことはほとんどなく，もっぱら compress が処理した圧縮ファイルの拡張子に用いられることが多いからです。

1)「*X680x0 libc*」Vol.2 P.315 を参照してください。

2)「*X680x0 libc*」Vol.2 P.7 を参照してください。

● fread 関数

テキストモードでオープンしたファイルからブロックの読み込みを行う場合，CR，LF シーケンスを LF コードへ置き換えます。従来 fread 関数[3]の戻り値は，この置き換えによって想定されない値を取ることがありましたが，これを戻り値には影響しないように変更しました。すなわち，正常に読み込めた場合は必ず指定したサイズのデータが読み込まれます。この仕様変更は ***MS-C 7.0*** との互換性を高め，正しくテキストモードを扱えるようにするためです。

3)「*X680x0 libc*」Vol.2 P.114 を参照してください。

● free 関数

従来 free 関数[4]は，解放するメモリブロックへのポインタに NULL ポインタを与えるとエラーとしていましたが，***ANSI C*** の規定によると NULL ポインタに対しては何もしないのが正しいようです。そのように仕様変更しました。

4)「*X680x0 libc*」Vol.2 P.115 を参照してください。

● fwrite 関数

テキストモードでオープンしたファイルにブロックの書き込みを行うとき，LF コードを CR，LF シーケンスへ置き換えます。従来 fwrite 関数[5]の戻り値は，この置き換えによって想定されない値を取ることがありましたが，これを戻り値には影響しないように変更しました。この仕様変更は ***MS-C 7.0*** との互換性を高め，正しくテキストモードを扱えるようにするためです。

5)「*X680x0 libc*」Vol.2 P.128 を参照してください。

● getcwd/getdcwd 関数

従来 getcwd 関数と getdcwd 関数[6]は，カレントワーキングディレクトリを格納するバッファへのポインタに NULL ポインタを与えるとエラーになっていましたが，NULL ポインタに対しては領域を関数の側で自動的に割り当てるように変更しました。この仕様変更は ***MS-C 7.0*** との互換性を高めるためです。

6)「*X680x0 libc*」Vol.2 P.136〜137 を参照してください。

● main 関数

main 関数が起動されるときの第 2 引数 argv 配列の最初の要素 argv[0] には，起動されたコマンド自体の名前が入ります。**XC** や **LIBC** では，ここにプロセス管理ポインタから得たパス名とコマンド名を設定していました。現在の **LIBC** では，コマンドが **HUPAIR** [7]のインタフェイスで起動された場合は，**HUPAIR** された引数から得た argv[0] をそのまま使うようにしました。**HUPAIR** 以外[8]のインタフェイスで起動された場合は従来と同じです。

7)「*X680x0 libc*」Vol.1 P.107 および Chapter3「**X680x0 HLK** の拡張概要 (P.54)」を参照してください。

8) COMMAND.X などです

● read 関数

テキストモードでオープンしたファイルからデータの読み込みを行うとき，CR，LF コードを LF のシーケンスに置き換えます。従来 read 関数[9]の戻り値は，この変換には関係なく常に指定したバイト数を読み込むようにしていましたが，これを指定したよりも少ないバイト数で返ることもあるように変更しました。この仕様変更は ***MS-C 7.0*** との互換性を高め，正しくテキストモードを扱えるようにするためです。

9)「*X680x0 libc*」Vol.2 P.250 を参照してください。

● readdir 関数

readdir 関数[10]の dirent 構造体に d_size，d_mode，d_time メンバを追加しました。各メンバの意味は Table 6-1 のとおりです。

メンバ d_size は stat 関数が渡す stat 構造体の中のメンバ st_size [11]とは異なり，ディレクトリの場合は 0 になります。また，これらの新しいメンバは ***Project LIBC Group*** が独自に追加したものなので，他の処理系に

10)「*X680x0 libc*」Vol.2 P.251 を参照してください。

11)「*X680x0 libc*」Vol.2 P.315 を参照してください。

Table 6－1 ● dirent 構造体の新メンバ

新メンバ	意　味
off_t d_size	ファイルサイズ
mode_t d_mode	拡張 UNIX ファイルモード
time_t d_time	最終変更時間

は存在しません。他の処理系フォームへの移植性を考慮するならば使用しないほうがよいでしょう。

● stat/lstat/fstat 関数

DRIVE.X などのコマンドを用いて，ドライブ番号を交換してあるような状態で stat 関数[12]を使用すると，異なったドライブをアクセスしてしまうことがありました。これを **Human68k** 内部のドライブ配置テーブルを直接参照するようにして，正しく扱えるよう仕様変更しました。

従来は仮想ディレクトリ，仮想ドライブが多重に組み合わされた場合のパス展開[13]は 2 重までしか扱いませんでしたが，この制限を撤廃しました。また，マウントポイントにおける stat 構造体の情報を，st_ino，st_dev，st_size についてはマウント先のディレクトリから，st_mode，st_mtime についてはマウント元のディレクトリから求めるように仕様変更しました。

stat 構造体に新しく st_blksize メンバを追加しました (Table 6-2)。ただしサイズは SunOS 4.X に合わせて 8192 に固定しています[14]。

stat 構造体の st_mode の S_IEXEC フラグの生成条件から“.Z”の拡張子を除外しました。現実的に，“.Z”という拡張子の実行ファイルを扱うことはほとんどなく，もっぱら compress が処理した圧縮ファイルの拡張子に用いられることが多いからです[15]。

12)「*X680x0 libc*」 Vol.2 P.315 を参照してください。

13) おもに st_dev，st_ino の生成時に使用されます。

14) なおこのメンバの正確な定義は，私たちにもよくわかりません。

15) 拡張子を見て判断するのも，本来は勧められるべきことではありませんが…。

Table 6-2 ● stat 構造体の新メンバ

新メンバ	意　味
st_blksize	ファイルシステムの入出力に最適なブロックサイズ

● write 関数

テキストモードでオープンしたファイルにデータの書き込みを行うとき，LF コードを CR，LF のシーケンスに置き換えます。従来 write 関数[16]の戻り値は，この変換によって指定したサイズよりも大きくなることがありましたが，これを戻り値には影響しないように仕様変更しました。すなわち成功した場合の戻り値は，常に指定した書き込みサイズとなります。この仕様変更は ***MS-C 7.0*** との互換性を高め，正しくテキストモードを扱えるようにするためです。

16)「*X680x0 libc*」 Vol.2 P.396 を参照してください。

6-3 追加した機能

現バージョンの **LIBC** では他の処理系や **XC** との互換性をさらに高めるために，当初のバージョンにさらに多くの関数を追加しました。本セクションでは，それら追加した関数の仕様(マニュアル)を解説します。「*X68k Programming Series #2 X680x0 libc*」「同 *Manual Books*」のマニュアルと合わせて参照してください。

- **atow**　文字列を符号つき short 型整数に変換する
- **ftw**　ファイルツリーの探索
- **getclock**　システムクロックの取得
- **getwd**　カレントワーキングディレクトリの取得
- **_harderr**　TRAP14 によるクリティカルエラーの処理
- **movedata**　メモリ領域のコピー
- **movmem**　メモリ領域のコピー
- **pclose**　入出力用パイプストリームのクローズ(プロセス間)
- **popen**　入出力用パイプストリームのオープン(プロセス間)
- **repmem**　メモリ領域の複数回コピー
- **setclock**　システムクロックの設定
- **setmem**　メモリ領域を指定文字で埋める
- **sigsetmask**　現在のプロセスシグナルマスクの設定
- **stcgfe**　ファイル名(拡張子)の解析
- **stcgfn**　ファイル名(ノード名)の解析
- **strbpl**　ポインタ配列の作成
- **strcasecmp**　2つの文字列を大文字小文字を区別しないで比較する
- **strins**　文字列の挿入
- **strmfe**　与えられた要素からファイル名を構成する
- **strmfn**　パスの各要素からパス名を構成する
- **strmfp**　パスの各要素からパス名を構成する
- **strncasecmp**　2つの文字列を大文字小文字を区別しないで指定文字数だけ比較する
- **strsrt**　配列を ASCII コード順にソートする
- **swmem**　メモリ領域の交換

atow

用　　途 —— 文字列を符号つき short 型整数に変換する。

書　　式 ——

```
#include <stdlib.h>
short atow (const char *nptr);
```

解　　説 —— atow 関数は *nptr* で指定された文字列を符号つき short 型整数に変換する。変換は先頭の空白を無視し，null 文字に出会うか，数値に変換できない文字に出会うまで行われる。

戻 り 値 —— 変換した結果を返す。

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— atof，atoi，atol

ftw

用　途——ファイルツリーを探索する。

書　式——

```
#include <ftw.h>
int ftw (const char *path,
        int (*fn) (const char *name, const struct stat *st,
        int info), int ndirs);
```

解　説——ftw関数は*path*で指定されたディレクトリおよびその下に存在するディレクトリについてファイルツリーを探索し，発見されるすべてのエントリ（ファイルあるいはディレクトリ）について*fn*で指定された処理関数を呼び出す。

それぞれのエントリに対して処理関数*fn*が呼ばれる場合，*name*にはパス名，*st*にはそのエントリに対してlstatした結果，*info*にはそのエントリの種別がそれぞれ渡される。*info*に渡される値の意味は次のとおり。

- FTW_F　通常ファイル
- FTW_D　ディレクトリ
- FTW_SL　シンボリックリンクファイル
- FTW_VL　ボリュームファイル
- FTW_NS　lstatできない

*path*で指定したディレクトリよりも下にあるディレクトリについては*ndirs*で指定した深さまで，再帰的に降下して探索が行われる。ここで，探索を下のディレクトリに降下させない場合の*ndirs*の値は1である。また，*st*の構造についてはlstat関数を参照のこと。

なおファイルツリーの探索は，すべてのエントリが処理されるか，処理関数*fn*が0以外の値を返すか，何らかのエラーが発生すると終了される。

戻り値——すべてのエントリについて処理した場合は0を返す。もし処理関数*fn*が0以外の値を返して処理を中断した場合は，ftw関数はこの処理関数が返した0以外の値をそのまま返す。また，処理が失敗した場合は−1を返し，変数errnoにその原因を示すエラーコードを設定する。

- EACCES　アクセスできない要素があった
- ENOENT　*path*で指定したパスが存在しない
- ENOTDIR　*path*で指定したパスはディレクトリではない
- EINVAL　不正な*ndirs*を指定した

●ELOOP　　シンボリックリンクのネストが深すぎるか，またはループしている

規　　格 —— *XPG3, AES/OS, SYSV*

関連項目 —— closedir, lstat, opendir, readdir, stat

getclock

用　途 —— システムクロックを取得する。

書　式 —— #include <sys/timers.h>

```
int getclock (int clock_type, struct timespec *tp);
```

解　説 —— getclock 関数は *clock_type* で指定したタイプでシステムクロックを取得し，*tp* ポインタが指す timespec 構造体にコピーする。*clock_type* に指定できる値は次のとおり。

- TIMEOFDAY　　現在時刻
- UPTIME　　起動してからの経過時間

また，timespec 構造体は次のとおり。

```
struct timespec{
    unsigned long tv_sec;    /* 積算秒 */
    long tv_nsec;            /* ナノ秒単位の端数 */
};
```

戻り値 —— 正常に取得できた場合は 0 を返し，失敗した場合には −1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　　不正な *clock_type* を指定した

規　格 —— ***POSIX.1, AES/OS***

関連項目 —— setclock, time

G

getwd

用　途 — カレントワーキングディレクトリを取得する。

書　式 —
```
#include <unistd.h>
char *getwd (char *buff);
```

解　説 — getwd 関数は，カレントドライブのカレントワーキングディレクトリを取得し，*buff* が指す領域にコピーする。パス名は，すべて先頭にドライブ名を付加した絶対パス名の形で返される。

```
"A:/foo/bar/simple/directory"
```

戻り値 — 正常に取得できた場合は *buff* を返し，失敗した場合には NULL を返す。

規　格 — ***4.3BSD***

関連項目 — getcwd，getdcwd

_harderr

用　途 —— TRAP14 によるクリティカルエラーの処理を行う。

書　式 ——
```
#include <signal.h>
void _harderr (void (*func)(int mode, int code, int must));
void __volatile _hardresume (int mode);
void __volatile _hardretn (int dos_err_code);
```

解　説 —— _harderr 関数は，ハードウェアエラーに対するエラー処理関数として，*func* で指定されたハンドラを登録する。登録されたハンドラは，DOS コールの入出力処理中にハードウェアエラーが発生した場合に呼び出されるようになる。

_hardresume 関数は，_harderr 関数で登録されたハンドラがどのように DOS コールを終了させるべきかを *mode* によって制御する。*mode* に指定できるのは次のとおり。

- _HARDERR_ABORT　　プログラムをアボートさせる
- _HARDERR_FAIL　　DOS コールをエラー終了させる
- _HARDERR_RETRY　　DOS コールを再試行する
- _HARDERR_IGNORE　　DOS コールのエラーを無視する

ただし，エラー状態であるにもかかわらず DOS コールを正常終了させると危険な場合があるので，今のところ _HARDERR_IGNORE は _HARDERR_FAIL とほぼ同一に機能する。また，強制的に正常終了させたいのであれば，_hardretn 関数を用いれば実現できる。

_hardretn 関数は _hardresume 関数の低レベル関数であり，result の値を D0 レジスタに代入して DOS コールを終了する。

戻り値 —— なし

注　意 —— *MS-C 7.0* では dos.h に定義されるが，ここではあえて signal.h に定義した。LIBC の関数のなかには，DOS コールが多数呼ばれるような場合に高速化のためエラーチェックを省いている箇所がいくつかある。そのような場合は，_HARDERR_FAIL を発行しても，1 回で呼び出している関数が終了しない場合がある。

規　格 —— ***Project LIBC Group, MS-C 7.0***

H

関連項目 —— signal

ハンドラ —— エラーハンドラ本体はユーザが作成する。そのハンドラのプロトタイプは以下のように宣言する。

```
void handler (int mode, int code, int must);
```

- *mode*　シグナルハンドラ内のエラー分類コード (etype_t) が格納される。現在は TYPE_DOS のみである
- *code*　エラーの種別を示す値 (ecode_t) が格納される
- *must*　発生したエラーが _hardresume 関数において FAIL, RETRY, IGNORE のどの選択が可能かを示す値が格納される。それぞれ選択可能な場合には対応ビットが 1 になる。たとえば (must & _HARDERR_FAIL) が真であれば, _hardresume 関数で FAIL を選択することができる

エラーハンドラは return 文, _hardretn 関数, _hardresume 関数の 3 種類の方法により終了させることができる。ただし return 文の場合はプログラムをアボートすることになる。etype_t, ecode_t の定義やその値については <sys/xsignal.h> (インストールしたディレクトリのなかの "include/sys/xsignal.h") を参照のこと。

サンプル —— List 6 1 エラーハンドラのプロトタイプ

```
 1: void handler (int mode, int code, int must)
 2: {
 3:     static const char *errmsg[] = {
 4:         "存在しないユニット番号を指定した",
 5:         "ドライブの準備ができていない",
 6:         "無効なコマンドである",
 7:         "データに CRC エラーが発生した",
 8:         "ディスクの管理領域が壊れている",
 9:         "シークエラーが発生した",
10:         "無効なメディア",
11:         "セクタが見つからない",
12:         "プリンタオフライン",
13:         "書き込みに失敗した",
14:         "読み出しに失敗した",
15:         "何だか知らないエラー",
16:         "ライトプロテクト",
17:         "書き込み不可(メディアが入ってない)"
18:     };
19:     fprintf (stderr, "%s: %s %s %s¥n",
```

```
20:                errmsg[errcode], "中止[A]",
21:           (must & _HARDERR_RETRY)  ? "再試行[R]" : "",
22:           (must & _HARDERR_IGNORE) ? "無視[I]"   : "",
23:           (must & _HARDERR_FAIL)   ? "失敗[F]"   : "");
24:       switch (toupper (getch ())) {
25:       case 'R':
26:          if (must & _HARDERR_RETRY)
27:               _hardresume (_HARDERR_RETRY);      /* 再試行 */
28:           break;
29:       case 'I':
30:           if (must & _HARDERR_IGNORE)
31:               _hardresume (_HARDERR_IGNORE);     /* 無視 */
32:           break;
33:       case 'F':
34:           if (must & _HARDERR_FAIL)
35:               _hardresume (_HARDERR_FAIL);       /* 失敗 */
36:           break;
37:       case 'A':
38:           default:
39:               break;
40:       }
41:       _hardresume (_HARDERR_ABORT);              /* アボート */
42: }
```

H

movedata

用　途 —— メモリ領域をコピーする。

書　式 ——
```
#include <string.h>
void *movedata (const void *region2, void *region1, size_t n);
```

解　説 —— movedata 関数は，*region2* の指す領域から *n* バイトを *region1* の指す領域にコピーする。movedata 関数は strcpy 関数とは異なり，null 文字を検出しても処理を中断しない。

戻り値 —— *region1* へのポインタを返す。

注　意 —— 領域が重なっていた場合の動作は未定義である。また *region1* は，*n* バイトのデータを格納するのに十分な領域を指していなければならない。movedata 関数と memmove 関数とは引数の位置が異なる。1 文字は 1 バイトとなる。

規　格 —— ***Project LIBC Group, XC***

関連項目 —— memcpy, memmove, movmem, setmem, strcpy, strncpy

movmem

用　　途——メモリ領域をコピーする。

書　　式——#include <string.h>

```
void *movmem (const void *region2, void *region1, size_t n);
```

解　　説——movmem 関数は *region2* の指す領域から *n* バイトを，*region1* の指す領域にコピーする。movmem 関数は strcpy 関数とは異なり，null 文字を検出しても処理を中断しない。また memcpy 関数とも異なり，*region1* と *region2* の指す領域が重なっていても正しくコピーできる。

戻 り 値——*region1* へのポインタを返す。

注　　意——*region1* は，*n* バイトのデータを格納するのに十分な領域を指していなければならない。movmem 関数と memmove 関数とは引数の位置が異なる。1 文字は 1 バイトとなる。

規　　格——***Project LIBC Group, XC***

関連項目——memcpy, memmove, movedata, setmem, strcpy, strncpy

M

pclose

用　途 —— プロセスとの間の入出力用パイプストリームをクローズする。

書　式 ——

```
#include <stdio.h>
int pclose (FILE *stream);
```

解　説 —— pclose 関数は *stream* が指すパイプストリームをクローズする。また，パイプストリームが書き込みモードでオープンされている場合は，書き込んだデータをもとに popen 関数で指定したコマンド文字列を実行する。

戻り値 —— 正常にクローズできた場合は 0 を返し，失敗した場合には −1 を返して，変数 errno にその原因を示すエラーコードを設定する。

●ENOMEM　　メモリが足りなくなった

注　意 —— pclose するパイプストリームは，必ず popen 関数で作成されていなければならない。通常の fopen 関数でオープンしたストリームを渡さないこと。

Human68k にはパイプという概念がないので，テンポラリファイルを用いた疑似パイプを利用する。このテンポラリファイルは，pclose 関数の処理が完了すると自動的に削除される。

Human68k はシングルタスクなので，常にプロセスを子プロセスとして実行しなければならない。したがって読み込みモードでオープンした場合は，パイプストリームから最初の読み込みを行う前に，すでにコマンドの実行は終了している。また逆に書き込みモードでオープンした場合，実際にコマンドが実行されるのは pclose した後である。

規　格 —— ***Project LIBC Group, AES, POSIX.1, SVID, 4.3BSD***

関連項目 —— fclose, fopen, popen

popen

用　途——プロセスとの間の入出力用パイプストリームをオープンする。

書　式——

```
#include <stdio.h>
FILE* popen (const char *command, const char *mode);
```

解　説——popen 関数は *command* で指される領域に指定されたコマンド文字列を実行し，そのプロセスとの間に入出力用のパイプストリームを作成する。パイプストリームが書き込みモードでオープンされていれば，自プロセスは起動したプロセスの標準入力に対してパイプストリームを通してデータを与えることができるし，読み込みモードでオープンされていれば，起動したプロセスの標準出力／エラー出力をパイプストリームを通して得ることができる。作成するパイプストリームのモードは *mode* が指す文字列で指定する。指定できるモード文字列は次のとおり。

- "r"　読み込み
- "w"　書き込み

また，モードを表す文字列の最後の次の文字を指定することで，パイプストリームのモードをテキスト／バイナリモードのどちらかにすることができる。省略時は，テキストモードあるいは fmode 関数で指定したモードに合わせられる。

- "b"　バイナリモード
- "t"　テキストモード (CRLF → LF 変換)

戻り値——正常にオープンできた場合はパイプストリームへのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　*command* で指定したコマンドが実行できない
- EINVAL　*mode* で不正なモードを指定した
- ENOMEM　メモリが足りなくなった

注　意——popen 関数で作成したパイプストリームは，必ず pclose 関数によってクローズされなければならない。通常の fclose 関数ではクローズしないこと。

Human68k にはパイプという概念がないので，テンポラリファイルを用

P

いた疑似パイプを利用する。したがって，**popen** 関数を実行するとテンポラリディレクトリにファイルを作成する。

Human68k はシングルタスクなので，常にプロセスを子プロセスとして実行しなければならない。したがって読み込みモードでオープンした場合は，パイプストリームから最初の読み込みを行う前にすでにコマンドの実行は終了している。また逆に書き込みモードでオープンした場合，実際にコマンドが実行されるのは pclose した後である。

規　　格 —— ***Project LIBC Group, POSIX.1, XPG3, AES/OS, 4.3BSD, SYSV***

関連項目 —— fclose, fopen, pclose

repmem

用　途——メモリ領域を複数回コピーする。

書　式——
```
#include <string.h>
void repmem (void *region1, const void *region2, size_t size,
             size_t n);
```

解　説——repmem 関数は，*region2* の指す領域から *size* バイトを *region1* の指す領域に *n* 回連続コピーする。repmem 関数は strcpy 関数とは異なり，null 文字を検出しても処理を中断しない。

戻り値——なし

注　意——領域が重なっていた場合の動作は未定である。また，*region1* は ($size \times n$) バイトのデータを格納するのに十分な領域を指していなければならない。

規　格——***Project LIBC Group, XC***

関連項目——memcpy

R

setclock

用　途 —— システムクロックを設定する。

書　式 ——

```
#include <sys/timers.h>
int setclock (int clock_type, struct timespec *tp);
```

解　説 —— setclock 関数は *clock_type* で指定したシステムクロックを *tp* ポインタで指される構造体 *timespec* の値で再設定する。ただし，*clock_type* に指定できる値は TIMEOFDAY だけである。

- ●TIMEOFDAY　　現在時刻

tp に指定する timespec 構造体は次のとおり。

```
struct timespec {
    unsigned long tv_sec;  /* 積算秒 */
    long tv_nsec;          /* ナノ秒単位の端数 */
};
```

戻り値 —— 正常に設定できた場合は 0 を返し，失敗した場合は −1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ●EINVAL　　不正な *clock_type* を指定した

規　格 —— ***POSIX.1*，*AES/OS***

関連項目 —— getclock，time

setmem

用　途 —— メモリ領域を指定文字で埋める。

書　式 —— `#include <string.h>`
`void *setmem (void *region, size_t n, int character);`

解　説 —— setmem 関数は，*region* の指す領域から *n* バイトを *character* で指定された文字で埋める。setmem 関数は strset 関数とは異なり，null 文字を検出しても処理を中断しない。*character* は，内部で int 型から unsigned char 型に変換される。

戻り値 —— *region* へのポインタを返す。

注　意 —— setmem 関数と memset 関数とは引数の位置が異なる。1 文字は 1 バイトとなる。

規　格 —— ***Project LIBC Group, XC***

関連項目 —— memchr, memcmp, memcpy, memmove, memset, movmem

sigsetmask

用　途 —— 現在のプロセスシグナルマスクを設定する。

書　式 ——
```
#include <signal.h>
int sigsetmask (int mask);
```

解　説 —— sigsetmask 関数は，現在ブロックされているシグナルのセットを *mask* の値で再設定する。発生したシグナルは，*mask* にそのシグナルに応じたビットが設定されているとブロックされる。

戻り値 —— 正常に設定できた場合は前のシグナルマスクを返し，失敗した場合は −1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　　不正な *mask* を指定した

注　意 —— sigsetmask 関数は古い形のインタフェイスである。したがって，引数や戻り値は sigset_t 型ではなく int 型となっている。**LIBC** では sigset_t 型は int 型と同じであるから問題ないが，移植性を考慮するならば使用すべきではない。

規　格 —— ***4.3BSD***

関連項目 —— sigblock, sigprocmask

stcgfe

用　　途 —— ファイル名 (拡張子) の解析を行う。

書　　式 ——
```
#include <string.h>
int stcgfe (char *ext, const char *name);
```

解　　説 —— stcgfe 関数は *name* が指すファイル名文字列から拡張子を取り出し，*ext* が指す領域に格納してその文字数を返す。

戻 り 値 —— 拡張子の文字数を返す。

注　　意 —— 1 文字は 1 バイトを意味する。

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— stcgfn

stcgfn

用　　途 —— ファイル名 (ノード名) の解析を行う。

書　　式 —— `#include <string.h>`

```
int stcgfn (char *node, const char *name);
```

解　　説 —— stcgfn 関数は *name* が指すファイル名文字列からノード ("."+ 拡張子を取り除いたもの) を取り出し，*node* が指す領域に格納してその文字数を返す。

戻 り 値 —— ノードの文字数を返す。

注　　意 —— 1 文字は 1 バイトを意味する。

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— stcgfe

S

strbpl

用　途 —— ポインタ配列を作成する。

書　式 ——

```
#include <string.h>
int strbpl (char **s, int max, const char *t);
```

解　説 —— strbpl 関数は *t* が指す文字列の並びをもとに，ポインタ配列を *s* に作成する。*t* はそれぞれ null 文字で終了する文字列の並びであり，*t* の最後にはさらにもう 1 つ null 文字が必要である。*max* には配列 *s* の要素数の最大値を指定する。

戻り値 —— ポインタの項目数を返す。*t* に含まれる要素数が *max* より大きい場合は −1 を返す。

注　意 —— 1 文字は 1 バイトを意味する。

規　格 —— ***Project LIBC Group, XC***

S

strcasecmp

用　途 —— 2 つの文字列を大文字と小文字を区別しないで比較する。

書　式 ——
```
#include <string.h>
int strcasecmp (const char *string1, const char *string2);
```

解　説 —— strcasecmp 関数は，*string1* の指す文字列と *string2* の指す文字列を大文字と小文字を区別しないで比較する。比較は null 文字が検出された時点で終了する。

戻り値 —— 比較の結果，2 つの文字列がまったく同じならば 0 を返す。異なる場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　意 —— strcasecmp 関数は文字列を小文字にしてから比較する。1 文字は 1 バイトを意味する。

規　格 —— ***Project LIBC Group, 4.3BSD***

関連項目 —— strcmpi，stricmp，strncasecmp

S

strins

用　　途 —— 文字列を挿入する。

書　　式 —— #include <string.h>

```
void strins (const char *src, char *dst);
```

解　　説 —— strins 関数は *dst* が指す文字列の前に，*src* が指す文字列を挿入し，1つの文字列にする。

戻 り 値 —— なし

注　　意 —— *dst* は *src* を挿入し，格納するのに十分な領域を指していなければならない。

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— strcat，strcpy

S

strmfe

用　　途 —— 与えられた要素からファイル名を構成する。

書　　式 ——
```
#include <string.h>
void strmfe (char *new, const char *old, const char *ext);
```

解　　説 —— strmfe 関数は *old* が指すファイル名文字列の拡張子を，*ext* が指す拡張子に置き換え，*new* が指す領域に格納する。

戻 り 値 —— なし

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— _makepath, _splitpath, strmfn, strmfp

strmfn

用　途 —— パスの各要素からパス名を構成する。

書　式 ——

```
#include <string.h>
void strmfn (char *path, const char *drive, const char *dir,
                const char *fname, const char *ext);
```

解　説 —— strmfn 関数は引数で指定された各要素から完全なパス名を構成し，その結果を *path* が指す領域に格納する。指定するそれぞれの引数の意味は次のとおり。

- *drive*
  ドライブ名。たとえば "A:" など。もしドライブ名の区切り記号であるコロンがない場合は自動的に加えられ，*drive* が NULL の場合は省略される

- *dir*
  ディレクトリ名。たとえば "foo/bar/" など。もし *dir* の最後にパスの区切り記号である "/" や "¥" がない場合は自動的に加えられ，*dir* が NULL の場合は省略される

- *fname*
  ファイル名。たとえば "sample" など拡張子を除いた部分。もし *fname* が NULL の場合は省略される

- *ext*
  ファイルの拡張子。たとえば ".c" など。もし先頭にピリオドがない場合は自動的に加えられ，*ext* が NULL ならば拡張子とピリオドは省略される

戻り値 —— なし

規　格 —— ***Project LIBC Group, XC***

関連項目 —— _makepath, _splitpath, strmfe, strmfp

S

strmfp

用　　途 —— パスの各要素からパス名を構成する。

書　　式 ——

```
#include <string.h>
void strmfp (char *path, const char *dir, const char *fname);
```

解　　説 —— strmfp 関数は引数で指定された各要素から完全なパス名を構成し，その結果を *path* が指す領域に格納する。指定するそれぞれの引数の意味は次のとおり。

● *dir*

ドライブ名を含むディレクトリ名。たとえば “a:/foo/bar/” など。もし *dir* の最後にパスの区切り記号である “/” や “¥” がない場合は自動的に加えられ，*dir* が NULL の場合も “/” を加える

● *fname*

ファイル名。もし *fname* が NULL の場合は省略される

戻 り 値 —— なし

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— _makepath, _splitpath, strmfe, strmfn

S

strncasecmp

用　途 —— 2つの文字列を大文字と小文字を区別しないで，指定文字数だけ比較する。

書　式 ——
```
#include <string.h>
int strncasecmp (const char *string1, const char *string2,
                 size_t n);
```

解　説 —— strncasecmp関数は，*string1*の指す文字列と*string2*の指す文字列を大文字と小文字を区別しないで比較する。比較はnull文字が検出されるか，*n*文字比較した時点で終了する。

戻り値 —— 比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での*string1*側の文字が*string2*側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　意 —— strncasecmp関数は文字列を小文字にしてから比較する。1文字は1バイトを意味する。

規　格 —— ***Project LIBC Group, 4.3BSD***

関連項目 —— strcasecmp, strnicmp

S

strsrt

用　　途 —— 配列を ASCII コード順にソートする。

書　　式 ——
```
#include <string.h>
void strsrt (char **s, int n);
```

解　　説 —— strsrt 関数は，*s* が指す文字列へのポインタの配列を ASCII コード順にソートする。*n* には配列の要素数を指定する。

戻 り 値 —— なし

注　　意 —— 1 文字は 1 バイトを意味する。

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— qsort

swmem

用　　途 —— メモリ領域を交換する。

書　　式 —— `void swmem (void *region1, void *region2, size_t n);`

解　　説 —— swmem 関数は，*region1* の指す領域と *region2* の指す領域を *n* バイト交換する。swmem 関数は null 文字を検出しても処理を中断しない。

戻 り 値 —— なし

規　　格 —— ***Project LIBC Group, XC***

関連項目 —— memcpy, memmove

6-4 ライブラリの互換性

「*X68k Programming Series #2 X680x0 libc*」に収録されたバージョン 1.0.20 と本書付属の最新版であるバージョン 1.1.31 の間の互換性は，構造体レベルの変更やヘッダファイルに定義されているマクロが変更されているため，バイナリレベルでは損なわれていますが，ソースレベルでは維持されています。ただし，すでに述べたように仕様変更があった部分については，それぞれソースレベルにおいても互換性がない部分がありますので注意が必要です。

本セクションでは，そうした非互換の部分について主だったポイント，特に注意しなければならない部分について解説します。

6-4-1 バイナリレベルの互換性

古い **LIBC** のヘッダファイルを用いてコンパイルしたその他のライブラリは，構造体やマクロが変更されたことにより互換性を失うことがあります[1)]。これらの互換性を失ったライブラリと最新版の **LIBC** を同時にリンクすると，実行時に暴走する可能性があります。必ず，再コンパイルしてから使用してください。

同様に，古い **LIBC** のヘッダファイルを用いてコンパイルしたオブジェクトは，互換性を失うことがあり，新しい **LIBC** とリンクすると暴走する可能性があります。必ず再コンパイルしてください。

1) 実際に失うかどうかは，使用している構造体や関数，マクロの種類によります。

6-4-2 ソースレベルの互換性

仕様変更された関数については，すでに述べた通りの非互換性があります[2)]。それ以外に次のような非互換部分があります。

2) 正しい形に変更しただけなので問題はないと思います。

●ライブラリの内部エラー／シグナル受信時の終了コード

ライブラリの内部エラーが発生したり，シグナルを受信してプロセスが異常終了する場合の終了コードが変わっています。従来は −254 などの適当な値を使用していましたが，現在のバージョンでは `wait` 構造体に合わせて決められています。もしこれらの終了コードを参照していた場合は，修正す

る必要があります。

● 構造体メンバの変更

LIBC は C++ コンパイラでも使用できるようになっています。これに加えて現在の LIBC ではその対応をより完全なものにし，さらに Objective-C でも使用できるようにしました。そのため，これら C++ や Objective-C の予約語[3]と衝突していた構造体メンバは名前を変更しています。したがって，これらのメンバを使用していた場合は変更が必要です。なお，Objective-C に関してはチェックしていません。

3) id, new, class などの予約語です。

Chapter 7

LIBC 便利帳

LIBC は *ANSI C* や *POSIX* などの標準規格になるべく準拠するように作成されており，使用する場合にも当然その規格からはずれないようにすべきです。しかし，X680x0 固有の機能を利用する (DOS コール，IOCS コールあるいはハードウェア直接) ことも当然あります。また，ライブラリの内部情報に直接アクセスして情報を取り出すことも (勧められませんが) 場合によっては有効でしょう。
本章では知っておくと便利なライブラリの内部情報について，その一部ですが掲載し，解説を加えておきたいと思います。ただし，これらの内部情報を利用する場合は，バージョンが変わったときに互換性を失う可能性があります。また，当然移植性を考慮するならば使用してはなりません。その点に注意してください。

7-1 プロセスメモリマップ

Human68k では，プログラムを起動するとそのプログラムのためにメモリブロックを割り当て，「メモリ管理ポインタ」，「プロセス管理ポインタ」などを作成し，そこにプログラムやデータをロードしていきます。しかし，C 言語で作成されたプログラムは，それ以外にも引数や環境変数エリアなどさまざまなデータを必要とします。

LIBC をリンクして作成されたプログラムも，もちろん起動時にスタートアップルーチンがこれらのデータ領域の割り当てを自動的に行います。本セクションでは，その各種データ領域の割り当て方 (プロセスメモリマップ) について解説していきます。

Fig. 7-1 は，**LIBC** がシステムから与えられたメモリをどのように区切って使用するかを図示したものです。この図中の各セクションバッファについて，それぞれ解説していきます。

◆ メモリ管理ポインタ (A)

Human68k 上で動作するプログラムは，基本的に **DOS** から割り当てられた以外のメモリをアクセスしてはなりません[1]。このメモリ管理ポインタは起動したプロセスのために割り当てられたメモリブロックを管理するための情報で，**Human68k** が使用しています。ここで，プロセスに割り当てられたメモリブロックとは Fig. 7-1 全体を指します。詳細については DOS コールである `_dos_malloc( )`，`_dos_malloc2( )` [2]を参照してください。この領域の先頭アドレスは変数 `_memcp` が指します。

◆ プロセス管理ポインタ (B)

プロセス管理ポインタは，起動したプロセスを管理するために必要な情報が納められており，**Human68k** が使用しています。この領域の先頭アドレスは変数 `_procp` が指します。

◆ テキスト (プログラムコード) セクション (C)

テキストセクションには，プロセスの実行部分のコードが格納されます。また，それ以外にも文字列定数や const 属性のデータもここに格納されることがあります[3]。この領域の先頭アドレスは変数 `_psta` が指します。

1) VRAM，I/O などを除きます。

2) 「*X680x0 libc*」 Vol.2 P.488～489 を参照してください。

3) 詳しくはコンパイラの方針によります。

領域	ポインタ
(A) メモリ管理ポインタ	_memcp
(B) プロセス管理ポインタ	_procp
(C) テキスト (プログラムコード) セクション	_psta
(D) 初期化済みデータセクション	_dsta
(E) 未初期化データセクション	_bsta
(F) コマンドラインバッファ	_csta
(G) 引数ベクタバッファ	_vsta
(H) 環境変数バッファ	_esta
(I) 環境変数ベクタバッファ	_fsta
(J) スタックエリア	_ssta
(K) ヒープエリア	_hsta

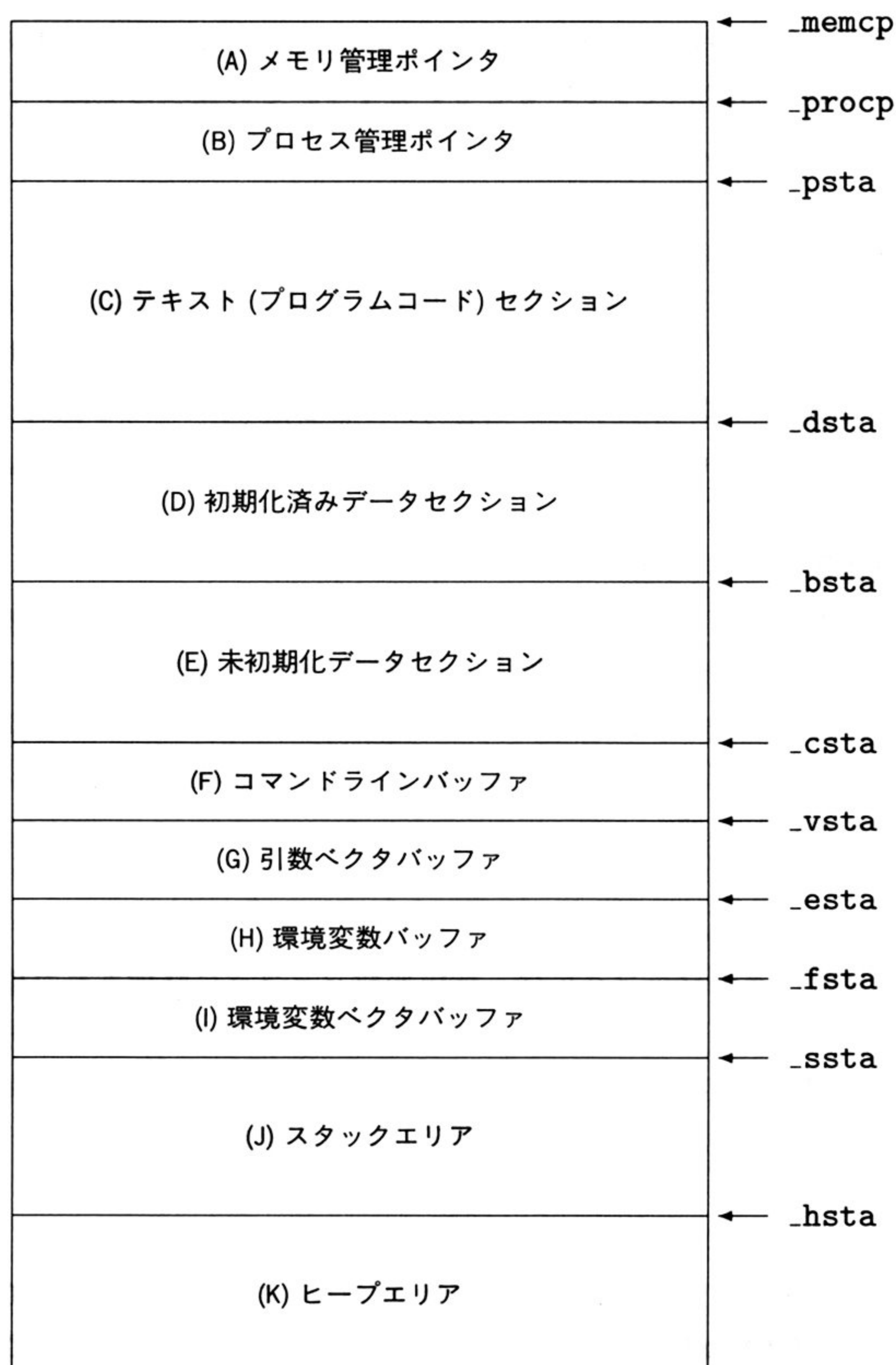



Fig. 7－1 ● プロセスメモリマップ

◆初期化済みデータセクション (D)

初期化済みデータセクションには，初期化データが指定された大域変数[4)]が格納されます。この領域の先頭アドレスは変数 _dsta が指します。

4) たとえば，"int a = 1;" といったものです。

◆未初期化データセクション (E)

未初期化データ (ブロックストレージ) セクションには，初期化要素のない大域変数[5)] が格納されます。この領域の先頭アドレスは変数 _bsta が指します。

5) たとえば "int a;" といったものです。

◆コマンドラインバッファ(F)

コマンドラインバッファには，プロセスを起動するときにユーザが指定したコマンドラインの内容が，引数ごとに NULL キャラクタで区切られて格

納されます。別のいい方をすれば，main 関数の第 2 引数である argv[] の各要素の実体が入っているともいえます。この領域の先頭アドレスは変数 _csta が指します。

◆引数ベクタバッファ(G)

引数ベクタバッファには，コマンドラインバッファに格納された引数列の各要素へのポインタが配列として格納されています。別のいい方をすれば main 関数の第 2 引数である argv[] の各要素 (ポインタ) が入っているともいえます。この領域の先頭アドレスは変数 _vsta が指します。

◆環境変数バッファ(H)

環境変数バッファには，プロセスが起動されたときの親プロセスのもっていた環境変数の情報が，引数ごとに NULL キャラクタで区切られて格納されます。別のいい方をすれば，main 関数の第 3 引数である environ[] の各要素の実体が入っているともいえます。この領域の先頭アドレスは変数 _esta が指します。

◆環境変数ベクタバッファ(I)

環境変数ベクタバッファには，環境変数バッファに格納された環境変数の各要素へのポインタが配列として格納されています。別のいい方をすれば main 関数の第 3 引数である environ[] の各要素 (ポインタ) が入っているともいえます。この領域の先頭アドレスは変数 _fsta が指します。

◆スタックエリア (J)

スタックエリアは，プロセスが実行時に使用するスタックとして使用されます。**MC680x0** ではスタックは下位アドレスに向かって消費されますから，実際にはこのスタックエリアはいちばん最後から順次使用されることになります。そして，いちばん先頭を越えてしまった状態をスタックオーバーフローと呼びます。この場合，関係のない領域がどんどん破壊され，さらに悪くなれば，実行コード部分まで破壊され暴走してしまいます。

スタックエリアは，前の環境変数ベクタバッファと後ろの ヒープエリアに挟まれた領域なので，足りなくなっても動的に増やすことができません。スタックをたくさん消費するようなプログラムは，意識的にこの領域を多めに確保する必要があるでしょう。なお，初期状態ではこのスタックエリアは 32K バイト確保されます。

このスタックエリアを 32K バイト以外の任意のサイズで確保するには，大域変数 _stacksize に必要なサイズ[6]を代入するだけでかまいません。**LIBC** は起動時にこの変数を参照して，その大きさを決めるからです。なお，この領域の先頭アドレス[7]は変数 _ssta が指します。

6) バイト単位で指定します。

7) つまりスタック後端のことです。

◆ **ヒープエリア (K)**

ヒープエリアはプロセスが実行時に自由に使用できるメモリ領域で，通常は，

malloc 関数[8]
realloc 関数[9]
free 関数[10]

などの関数を用いて操作します。このヒープエリアは後ろに何もないので，足りなくなった場合は自動的にサイズが拡張されます。これは，**Human68k** から自プロセスに割り当てられたメモリブロックのサイズを大きくすることで実現します。ただし，プロセスの後ろに常駐ソフトウェアなどがあり，サイズを大きくすることができない場合もあります。なお，この領域の先頭アドレスは変数 _hsta が指し，後端は変数 _last が指します。

8)「*X680x0 libc*」Vol.2 P.207 を参照してください。
9)「*X680x0 libc*」Vol.2 P.253 を参照してください。
10)「*X680x0 libc*」Vol.2 P.115 を参照してください。

7-1-1 メモリに関する注意点

LIBC の malloc 関数，realloc 関数でメモリを確保するためには，DOS コール _dos_setblock 関数[11]で取得できる連続したプロセスメモリブロックが必要になります。そのため，_dos_malloc 関数[12]などの DOS コールで，直接 **Human68k** からメモリを確保するなどしてメモリブロックを分断した場合には，メモリに十分なフリーエリアが存在するにもかかわらず，ヒープエリアを拡張できなくなるため，malloc 関数や realloc 関数が失敗することがあります。これを回避するためには，次のような方法があります。

11)「*X680x0 libc*」Vol.2 P.518 を参照してください。
12)「*X680x0 libc*」Vol.2 P.488 を参照してください。

- **LIBC が管理する malloc 関数と Human68k が管理する _dos_malloc 関数を同じプログラム内で混在して使わない**
- **あらかじめ必要とする十分なヒープエリアを確保しておく[13]**
- **_dos_malloc 関数と _dos_mfree 関数[14]を対にして使用し，malloc する前に必ず _dos_mfree することでメモリが分断しないようにする**

いずれにせよ，機能が同じだからといってレベルの異なる 2 種類の関数を同時に使用すると，いろいろな障害が発生します。なるべく，同じレベルの関数だけで完結するようにしてください[15]。

13)詳しくは「内部変数一覧」_heapsize (P.143) についての説明を参照してください。
14)「*X680x0 libc*」Vol.2 P.491 を参照してください。
15)たとえば，標準関数なら標準関数だけ，DOS コールなら DOS コールだけといったぐあいにです。

7-2 内部変数一覧

1) たとえば変数 errno などがそうです。

LIBC は「*X680x0 libc*」に掲載されている公開された大域変数[1] 以外にも，ライブラリ内部で参照している非公開の変数がたくさんあります。その一部を以下で解説します。いくつかの変数は，有用な場合もあるかもしれません。

◆ _argc

main 関数に渡される第 1 引数 argc が保存してあります。main 関数以外の関数から argc を参照したい場合に利用できます。

◆ _argv

main 関数に渡される第 2 引数 argv が保存してあります。main 関数以外の関数から argv を参照したい場合に利用できます。

◆ _bsta

未初期化データセクションの先頭アドレスが保存してあります。未初期化データセクションは，_bsta ～ (_csta - 1) の領域に位置しています。

◆ _comline

プロセスを起動する際に指定されたコマンドライン文字列へのポインタを保存してあります。独自の方法でコマンドラインを再度解析したい場合などに用いることができます。

◆ _cplusplus

C++ プログラムあるいは "-+-g" スイッチをつけて起動されたプログラムは，この値が 1 になります。自分が C++ プログラムであるかどうかを調べることができます。

◆ _dsta

初期化済みデータセクションの先頭アドレスが保存してあります。初期化済みデータセクションは，_dsta ～ (_bsta - 1) の領域に位置しています。

◆ _fddb

現在使用できるすべてのファイルハンドルについての詳細な情報が保存してあります。この構造体 _handles の配列をアクセスしてオープンしたファイルの絶対パス名や FCB の情報を知ることができます。_handles 構造体

の構造は次のとおりです。

List 7-1 _handles 構造体の内容

```
1: struct _handles {
2:     int inuse;                    /* ライブラリで使用中なら 0 以外の値 */
3:     int unlink;       /* クローズ後の削除が予約されていれば 0 以外の値 */
4:     int oflag;                    /* オープンモード */
5:     int atty;                     /* キャラクタデバイスならば 0 以外の値 */
6:     int iomd;                     /* デバイスモードの保存値 */
7:     union _fcb fcb;               /* オープン時の FCB の保存値 */
8:     char abspath[PATH_MAX + 1]; /* オープンファイルの絶対パス名 */
9:     char reserved[12];            /* 予約領域(未使用) */
10: };
```

◆ **_hsta**

ヒープエリアの先頭アドレスが保存してあります。ヒープエリアは，_hsta ～ (_last - 1) のエリアに位置しています。

◆ **_heapsize**

起動時のヒープエリアのサイズが保存されています。ただしこの値は起動時の値であり，現在の値ではありません。現在の値を知りたい場合は，(int)(_last - _hsta) を計算してください。

◆ **_hupair**

HUPAIR インタフェイスを通して起動された場合は，この値が 1 になります。自分がどういう方法で起動されたかを調べることができます。

◆ **_memcp**

メモリ管理ポインタの値が保存してあります。プログラム中から，直接メモリ管理ブロックを参照したい場合に利用できます。

◆ **_psta**

テキスト(プログラムコード)セクションの先頭アドレスが保存してあります。テキストセクションは，_psta ～ (_dsta - 1) の領域に位置しています。

◆ **_procp**

プロセス管理ポインタの値が保存してあります。プログラム中から，直接プロセス管理ブロックを参照したい場合に利用できます。

◆ **_ssta**

スタックエリアの先頭アドレスが保存してあります。スタックエリアは，_ssta ～ (_hsta - 1) のエリアに位置しています。

◆ _stacksize

起動時のスタックエリアのサイズが保存されています。このサイズは可変ではないので，常に一定となります。

7-3 起動オプションスイッチ一覧

LIBC を使用して作成されたプログラムを起動すると，起動時に指定した引数がそのまま引数の配列として main 関数に渡されます。アプリケーションはこの引数の配列を調べて，オプションスイッチやファイル名などを取り出すわけですが，これと同様に **LIBC** へのオプションスイッチというものが存在します。**LIBC** のオプションスイッチとは起動時に **LIBC** が解釈し，ライブラリ自体の動作を決定するためのものです。このオプションスイッチは **LIBC** に対するものなので，ライブラリが引数配列から抜き取って，main 関数には渡されません。そのために，このオプションスイッチはアプリケーションのオプションスイッチと区別するため[1]，やや冗長な形式となっています。現在，**LIBC** は次のライブラリオプションを解釈します。なお，これらのオプションスイッチは「*X680x0 libc*」に付属する **LIBC** のバージョンから変更されていません。

1) また，アプリケーションの自由度を制限しないためです。

- -+-s:*bytes*　スタックサイズの指定
- -+-h:*bytes*　ヒープサイズの指定
- -+-p　スーパーバイザモードへの移行
- -+-f　FLOATn.X を使用して数値計算を行う
- -+-g　C++ 用

-+-s:*bytes*　スタックサイズの指定

書式：　-+-s:*bytes*

機能：　スタックエリアのサイズを指定します。

解説：　*bytes* で指定したバイト数をスタックエリアに割り当てます。このオプションスイッチを指定しない場合，標準でスタックエリアは 32K バイト確保されます。-+-s: と *bytes* の間は空白を入れず，また *bytes* はバイト単位で指定してください。

-+-h:*bytes*　ヒープサイズの指定

書式：　-+-h:*bytes*

機能：　ヒープエリアのサイズを指定します。

解説：　*bytes*で指定したバイト数をヒープエリアに割り当てます。ヒープエリアは足りなくなった時点で自動的に拡張されるので，この値は初期サイズを指定するためのものと考えてください。このオプションスイッチを指定しない場合，標準でヒープエリアは64Kバイト確保されます。-+-h:と*bytes*の間は空白を入れず，また*bytes*はバイト単位で指定してください。

-+-p　スーパーバイザモードへの移行

書式：　-+-p

機能：　動作モードをスーパーバイザモードに変更します。

解説：　このオプションスイッチを指定すると，main関数の実行が開始される前にアプリケーションをスーパーバイザモードに変更します。その結果，I/Oコプロセッサを利用した数値演算が高速化されますが，当然，メモリ保護が効かなくなるなどの危険がともないます。このオプションスイッチを指定しない場合，アプリケーションはユーザモードで動作します。

-+-f　FLOATn.Xを使用して数値計算を行う

書式：　-+-f

機能：　数値演算にコプロセッサを使用します。

解説：　**LIBC**はコプロセッサが利用可能ならば，それらを用いて数値計算を行います。ただし，いろいろな理由でコプロセッサの代わりにソフトウェアで計算したい場合があります。そのような場合は，このオプションスイッチを指定してプログラムを起動してください。**LIBC**は数値演算をFLOATn.Xを使って行うようになります。

-+-g　C++ 用

書式：　-+-g

機能：　グローバルコンストラクタ/グローバルデストラクタを起動します。

解説：　C++ プログラムを開発する場合，libcplus.a [2]をリンクする必要があります。しかし，もしこのライブラリをリンクし忘れたり，意図的にリンクしたくない場合は，このオプションスイッチを指定することでプログラム中のグローバルコンストラクタ，グローバルデストラクタを強制的に起動することができます。

2) 「*X680x0 libc*」 Vol.1 P.32 を参照してください。

7-3-1 GCC の起動オプションスイッチと LIBC

メモリマップに関する解説[3]の部分で，すでにスタックエリア，ヒープエリアに関しては説明しました。また，これらのエリアの初期サイズを起動オプションスイッチや変数 _stacksize，変数 _heapsize で変更することができるということも説明しました。しかし，これらのエリアの初期サイズを変更する方法がもう 1 つあります。

X680x0 GCC には “-z-heap” と “-z-stack” スイッチで，コマンドラインから **XC** のスタックエリア，ヒープエリアのサイズが変更できます[4]。**LIBC** は **XC** と同じく，その変更機能が利用できるようになっています。必要な **GCC** のオプションスイッチは次のとおりです。

3) 「プロセスメモリマップ」(P.138) を参照してください。

4) 「*X68000 Develop.*」 Vol.2 P.44 を参照してください。

-z-heap=*bytes*　ヒープサイズの指定

書式：　-z-heap=*bytes*

機能：　生成実行ファイルのヒープサイズを指定します。

解説：　*bytes* で指定したバイト数をヒープエリアに割り当てます。ヒープエリアは足りなくなった時点で自動的に拡張されるので，この値は初期サイズを指定するためのものと考えてください。このオプションスイッチを指定しない場合，標準でヒープエリアは 64K バイト確保されます。*bytes* はバイト単位で指定してください。

-z-stack=*bytes*　スタックサイズの指定

書式：　-z-stack=*bytes*

機能：　生成実行ファイルのスタックサイズを指定します。

解説：　*bytes* で指定したバイト数をスタックエリアに割り当てます。このオプションスイッチを指定しない場合，標準でスタックエリアは 32K バイト確保されます。*bytes* はバイト単位で指定してください。

7-4 エラーメッセージ一覧

どうしても必要な場合，**LIBC** は次のようなエラーメッセージを表示します。このエラーメッセージはプログラムの動作状況によらず，常にコンソールに表示されます（リダイレクトの影響を受けません）。

◆スタックオーバーフロー

下記のエラーメッセージは，スタックオーバーフローが起こったことを知らせるメッセージです。**LIBC** はこのメッセージを表示した後，プログラムを強制終了させます。**X680x0 GCC** で -fstack-check スイッチ[1]を指定してコンパイルすると，スタックチェックつきの実行ファイルができあがります。この実行ファイルは関数呼び出しのたびに，スタックポインタがスタックエリアからはみ出していないかどうかをチェックします。もしスタックエリアをはみ出し，暴走する可能性がある場合はこのメッセージが表示されます。

1）「*X68000 Develop.*」Vol.2 P.39 を参照してください。

```
libc: stack overflow.
```

◆メモリブロック配置エラー

下記のエラーメッセージは，プログラムを実行するだけのメモリが足りない場合に表示されるメッセージです。**LIBC** はこのメッセージを表示した後，プログラムを強制終了させます。このメッセージはプログラムの実行に必要なスタックエリアとヒープエリアが確保できないことを意味していますが，ヒープエリアの拡張とは関係がありません。main 関数の処理が始まるよりも前に，プロセスのメモリマップを決定する段階で発生するエラーです。この段階ではアプリケーションは何も実行されていません。

```
libc: setblock failed.
```

7-5 エラーコード一覧

LIBC で定義されているエラーコードの一覧を次に掲載します。これらの値は関数実行時にエラーが発生したときに，その原因を示すために大域変数 errno に設定されるものです[1]。実際には，これらのエラーコードは **Human68k** が返すエラーコードから意味的，統計的に変換して得られるだけであり，必ずしも現実に即さない場合があります。したがって，ここでは各エラーコードのもつ基本的な意味だけを解説することにします。

なお，「*X68k Programming Series #2 X680x0 libc*」での公開時[2]には未使用となっていたいくつかのコードも，現在は使用しています。

1) 詳しくは実際に使用するか，関数のマニュアルを参照してください。

2) 「*X680x0 libc*」Vol.1 P.101 を参照してください。

- ●EDOM　算術関数の引数が定義域外である
- ●ERANGE　算術関数の結果が大きすぎる
- ●E2BIG　引数リストが長すぎる
- ●EACCES　指定したパス名にアクセスできない
- ●EAGAIN　リソースに一時的にアクセスできない
- ●EBADF　不正なファイルハンドルを指定した
- ●EBUSY　リソースが使用中である
- ●ECHILD　子プロセスが存在しない
- ●EDEADLK　デッドロックが起きてしまう
- ●EEXIST　指定したファイルがすでに存在している
- ●EFAULT　不正なアドレスを指定した
- ●EFBIG　ファイルが大きすぎる
- ●EINTR　関数の実行が割り込みによって中断された
- ●EINVAL　不正な引数を指定した
- ●EIO　入出力エラー
- ●EISDIR　指定したパス名がディレクトリである
- ●ELOOP　シンボリックリンクのネストが深すぎるか，ループしている
- ●EMFILE　これ以上ファイルをオープンできない
- ●EMLINK　これ以上ファイルをリンクできない
- ●ENAMETOOLONG　指定したパス名が長すぎる
- ●ENFILE　これ以上ファイルをオープンできない(システム全体)
- ●ENODEV　指定したデバイスが見つからない

- ●**ENOENT**　指定したパス名が見つからない
- ●**ENOEXEC**　不正な実行フォーマットである
- ●**ENOLCK**　これ以上ロックできない
- ●**ENOMEM**　メモリが足りない（ヒープエリアが足りない）
- ●**ENOSPC**　ディスクが一杯になった
- ●**ENOTBLK**　指定したデバイスはブロックデバイスではない
- ●**ENOSYS**　サポートしていない機能である
- ●**ENOTDIR**　指定したパス名はディレクトリではない
- ●**ENOTEMPTY**　指定したディレクトリが空ではない
- ●**ENOTTY**　不正な I/O 操作を行った
- ●**ENXIO**　指定したデバイス／アドレスが見つからない
- ●**EPERM**　禁止された操作を行おうとした
- ●**EPIPE**　パイプが壊れている
- ●**EROFS**　読み込み専用のファイルシステムである
- ●**ESPIPE**　パイプ上ではシークできない
- ●**ETXTBSY**　実行ファイルが使用中である
- ●**ESRCH**　指定したプロセスが見つからない
- ●**EXDEV**　ファイルシステムを越えてリンクしようとした
- ●**EDEVFS**　デバイスファイルシステムである
- ●**EWOULDBLOCK**　デッドロックが起きてしまう

7-6 環境変数一覧

LIBC の提供する関数群のなかには，環境変数を参照して動作を変化させるものがいくつかあります。現在，**LIBC** は次の環境変数を参照しています。なお，これらの環境変数はバージョン 1.0.20 から変更されていません。

◆ path

spawnlp 関数や execlp 関数など，外部プログラムを実行させる関数は環境変数 path を参照し，指定されたプログラムを path ディレクトリから検索します。また当然ですが，この環境変数は必ず設定しておく必要があります。

◆ temp

tmpnam 関数など，テンポラリファイルを作成する関数は環境変数 temp を参照し，指定されたディレクトリにテンポラリファイルを作成します。また，この環境変数は必ず設定しておく必要があります。

◆ USER, LOGNAME

LIBC では getlogin 関数など，ユーザ名／ユーザログイン名を取得する関数を提供していますが，**Human68k** にはこのような概念がありません。そこで **LIBC** はこれらのユーザ名を取得する場合，環境変数 USER, LOGNAME を参照します。値は環境変数 USER の設定が優先されますが，もし USER が未定義ならば，環境変数 LOGNAME を使用します。また，LOGNAME も未定義ならば固定的に “root” が使用されます。またこの環境変数は，必要がなければ指定する必要はありません。

◆ UID, EUID

LIBC では getuid 関数や geteuid 関数など，ユーザ ID ／実効ユーザ ID を取得する関数を提供していますが，**Human68k** にはこのような概念がありません。そこで **LIBC** は，これらの値を環境変数 UID, EUID から参照します。それぞれ UID がユーザ ID, EUID が実効ユーザ ID を表します。もし環境変数 EUID が未定義ならば，実効ユーザ ID はユーザ ID と等

しくなります。また環境変数 UID も未定義ならば，ユーザ ID は 0 (root) となります。また，この環境変数は，必要がなければ指定する必要はありません。

◆ GID, EGID

LIBC では getgid 関数や getegid 関数など，グループ ID／実効グループ ID を取得する関数を提供していますが，**Human68k** にはこのような概念がありません。そこで **LIBC** は，これらの値を環境変数 GID, EGID から参照します。それぞれ GID がグループ ID, EGID が実効グループ ID を表します。もし環境変数 EGID が未定義ならば，実効グループ ID はグループ ID と等しくなります。また環境変数 GID も未定義ならば，グループ ID は 0 (root) となります。また，この環境変数は，必要がなければ指定する必要はありません。

◆ SYSROOT

パスワードファイル (/etc/passwd), グループファイル (/etc/group) は名前が固定のファイルです。通常，これらのファイルは "A:/" から読み込まれますが，環境変数 SYSROOT を設定することで，読み込む位置を変更することができます。たとえば環境変数 SYSROOT に "B:/user" を設定すると，パスワードファイルは "B:/user/etc/passwd" が読み込まれます。またこの環境変数は，必要がなければ指定する必要はありません。

◆ SHELL, SYSTEM_SHELL

system 関数で，外部プログラムを起動する場合に使用するシェル[1]を指定します。もしこれが設定されていない場合，**LIBC** は COMMAND.X をシェルとして使用します。またこの環境変数は，必要がなければ指定する必要はありません。

1) コマンドインタプリタ，たとえば COMMAND.X などのことです。

◆ SHELL_OPT, SYSTEM_SHELL_OPT

system 関数で外部プログラムを起動する場合に，シェルに対して引数を渡すのに用いられるオプションスイッチを指定します。もしこれが設定されていない場合，**LIBC** は使用するシェルの形式によって自動的にオプションスイッチを選択します。またこの環境変数は，必要がなければ指定する必要はありません。

◆ SHELL_TYPE, SYSTEM_SHELL_TYPE

system 関数で外部プログラムを起動する場合に，使用するシェル (コマンドインタプリタ) のタイプを指定します。タイプは，**Human68k** の

COMMAND.X 形式のものと **UNIX** ライクないわゆる「シェル」形式のものと 2 種類あります。またこの環境変数は，必要がなければ指定する必要はありません。

◆ limit_core

LIBC では libsignal.a [2]を用いてシグナル機構をエミュレートしており，コアダンプも制限がない限り行います[3]。環境変数 limit_core はこのコアダンプの制限を指示するためのもので getrlimit 関数の RLIMIT_CORE [4] パラメータの値と関連づけられています。

この環境変数 limit_core が設定されていないか，その値が 0 の場合はコアダンプは禁止されます。また，−1 の場合はサイズ制限なくコアダンプが行われます。任意のサイズまででサイズを制限したい場合は，そのサイズをバイト単位で指定してください。

2) 「*X680x0 libc*」 Vol.1 P.35 を参照してください。

3) コアダンプについては，**signal** 関数（「*X680x0 libc*」Vol.2 P.289）の説明を参照してください。

4) 「*X680x0 libc*」 Vol.2 P.158 を参照してください。

7-7 シグナル一覧

LIBC では libsignal.a を用いてシグナル機構をある程度までエミュレートしており，たとえばバスエラーにより SIGSEGV シグナルが発生したり，0 除算により SIGFEP シグナルが発生したりします。これらのシグナルの扱いについてはすでに「*X68k Programming Series #2 X680x0 libc*」で詳しく説明したとおりですが[1)]，どのような場合にどのようなシグナルが発生するのかということまでは触れていませんでした。

そこで本セクションでは，**Human68k** および **X680x0** が実際に発生するエラーとそれにより発生するシグナルとがどのように結びついているかを，以下に示します。なお，この結びつきは Sun Micro Systems の Sun3 シリーズで使用されていた SunOS 4 を参考にして作成されました。

1) 「*X680x0 libc*」Vol.2 signal 関数 (P.289), sigaction 関数 (P.280) などを参照してください。

◆0 - Inquery signal

このシグナルは，kill 関数[2)]がソフトウェア的に生成するシグナルで，それ以外の状況では発生しません。シグナル 0 番は，普通シグナルを受けるプロセスが生きているかどうかを調べるためのものですが，**Human68k** はシングルタスクなのでとくに使用方法はありません。

2) 「*X680x0 libc*」 Vol.2 P.191 を参照してください。

◆SIGABRT - Abnormal program termination

このシグナルは，abort 関数[3)]がソフトウェア的に生成するシグナルで，実行中のプロセスを異常終了 (強制終了) させるために用いられます。

3) 「*X680x0 libc*」 Vol.2 P.5 を参照してください。

◆SIGFEP - Mathematical exception

このシグナルは，算術的なエラーの場合に生成されるシグナルです。たとえば,「0 による除算」,「不当な CHK 命令を実行しようとした」,「不当な TRAPV 命令を実行しようとした」などの場合が該当します。また，kill 関数によってソフトウェア的に生成することもできます。

◆SIGILL - Illegal instruction

このシグナルは，命令の実行エラーが発生したときに生成されます。たとえば,「MPU が不当な命令を実行しようとした」,「特権違反が起きた」などの場合が該当します。また，kill 関数によってソフトウェア的に生成する

こともできます。

◆SIGINT - Interrupted

このシグナルは，キーボードからの CTRL + C 割り込みが発生した場合にソフトウェア的に生成されます。発生させる原因は **Human68k** であり，その仕様から完全な割り込みにはなりえません。

◆SIGSEGV - Segmentation fault

このシグナルは，バスエラーが発生したときに生成されます。また，kill 関数によってソフトウェア的に生成することもできます。

◆SIGTERM - Terminated

このシグナルは kill 関数がソフトウェア的に生成するシグナルで，それ以外の状況では発生しません。SIGTERM は普通シグナルを受けるプロセスを停止させるときに用いられますが，**Human68k** はシングルタスクなので，とくに使用方法はありません。互換性のために存在します。

◆SIGALRM - Alarm clock

このシグナルは alarm 関数[4]によって設定された RTC(リアルタイムクロック) からのアラームが発生したときに生成されます。また，kill 関数によってソフトウェア的に生成することもできます。

4) 「*X680x0 libc*」 Vol.2 P.10 を参照してください。

◆SIGKILL - Killed

このシグナルは kill 関数がソフトウェア的に生成するシグナルで，それ以外の状況では発生しません。SIGKILL は普通シグナルを受けるプロセスを強制終了させるときに用いられますが，**Human68k** はシングルタスクなので，とくに使用方法はありません。互換性のために存在します。

◆SIGBUS - Bus error

このシグナルは，アドレスエラーが発生したときに生成されます。また，kill 関数によってソフトウェア的に生成することもできます。

◆SIGSTOP - Stopped

このシグナルは kill 関数がソフトウェア的に生成するシグナルで，それ以外の状況では発生しません。SIGSTOP は普通シグナルを受けるプロセスを suspend させるときに用いられますが，**Human68k** はシングルタスクなので，とくに使用方法はありません。互換性のために存在します。

◆ SIGEMT - EMT Trap

このシグナルは，未定義トラップが発生したときに生成されます。たとえば,「未定義割り込みが発生した」,「未定義の IOCS コールを実行しようとした」,「未定義の DOS コールを実行しようとした」,「未定義の F ライントラップが発生した」などの場合が該当します。また，kill 関数によってソフトウェア的に生成することもできます。

◆ SIGUSR1 - User defined 1

このシグナルは kill 関数がソフトウェア的に生成するシグナルで，それ以外の状況では発生しません。互換性のために存在します。

◆ SIGUSR2 - User defined 2

このシグナルは kill 関数がソフトウェア的に生成するシグナルで，それ以外の状況では発生しません。互換性のために存在します。

Chapter 8

Appendix A

本章では，大幅に拡張されたアドレス形式のアセンブリ言語での表記方法とアドレス形式指定をするうえでの注意点について解説します。

A-1 アドレス形式の表記

68020 以降の CPU では，**アドレス形式**が 68000，68010 から大幅に拡張されています。ここでは，拡張されたアドレス形式を含めたすべてのアドレス形式のアセンブリ言語での表記の方法について説明します。

A-1-1 データレジスタ直接形式

操作の対象は指定したデータレジスタ D*n* で，命令の操作は指定したレジスタの内容に対して直接実行されます。

書式： D*n* (*n* = 0～7)

A-1-2 アドレスレジスタ直接形式

操作の対象は指定したアドレスレジスタ A*n* で，命令の操作は指定したレジスタの内容に対して直接実行されます。

書式： A*n* (*n* = 0～7)

A-1-3 アドレスレジスタ間接形式

操作の対象は，指定したアドレスレジスタ A*n* の内容によってポイントされるメモリ内のデータです。

書式： (A*n*) (*n* = 0～7)

A-1-4 ポストインクリメント・アドレスレジスタ間接形式

操作の対象は，指定したアドレスレジスタ A*n* の内容によってポイントされるメモリ内のデータです。オペランドに対して命令の操作が実行された後に，指定のアドレスレジスタに命令のサイズ [1]が加算されます。

書式： (A*n*)＋ (*n* = 0～7)

1) “.b” なら 1，“.w” なら 2，“.l” なら 4 になります。

A-1-5 プリデクリメント・アドレスレジスタ間接形式

操作の対象は，指定したアドレスレジスタ An の内容によってポイントされるメモリ内のデータです。オペランドに対する命令の操作の実行に先立って，指定のアドレスレジスタから命令のサイズ[2)]が減算されます。

書式：　–(An)　(n = 0 ~ 7)

2) “.b” なら 1，“.w” なら 2，“.l” なら 4 になります

A-1-6 ディスプレースメントつきアドレスレジスタ間接形式

指定したアドレスレジスタ An の内容と，16 ビットのディスプレースメント値 d (−32768 ~ 32767) を 32 ビットに符号拡張した値を加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。

書式：　(d, An)　(n = 0 ~ 7)
d ：16 ビットのディスプレースメント (−32768 ~ 32767)

A-1-7 インデックスつきアドレスレジスタ間接形式

◆ **8 ビットディスプレースメント**

このアドレス形式は，68000，68010 のインデックスつきアドレスレジスタ間接形式の拡張になっています。68020 以降になって，インデックスレジスタに対するスケール値の指定が追加されました。

指定したアドレスレジスタ An の内容と，8 ビットのディスプレースメント値 d (−128 ~ 127) を 32 ビットに符号拡張した値，そしてインデックスとして指定したレジスタ Rn[3)]の内容とスケール値 *scale* との積をすべて加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。
インデックスレジスタ Rn には，ワードまたはロングワードのサイズ指定が必要です。ワードサイズを指定すると，レジスタの内容の下位 16 ビットを 32 ビットに符号拡張した値とスケール値との積がアドレス指定に使用されます。

3) データレジスタまたはアドレスレジスタです。

書式：　(d,An,Rn.*size***scale*)
d　：8 ビットのディスプレースメント (−128 ~ 127)
size　：インデックスレジスタのサイズ指定 (w または l)
scale　：スケール値指定 (1, 2, 4, 8 のいずれか)

◆ **ベースディスプレースメント**

このアドレス形式は，従来のインデックスつきアドレスレジスタ間接形式

のディスプレースメントを 16 または 32 ビットに広げる拡張が行われたものです。

指定したアドレスレジスタ A*n* の内容と，16 ビットまたは 32 ビットのベースディスプレースメント *bd*[4]，そしてインデックスとして指定したレジスタ R*n*[5] の内容とスケール値 *scale* との積をすべて加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。
このアドレス形式では，アドレスレジスタ，ベースディスプレースメント，インデックスレジスタのそれぞれを任意に省略することができ，省略した値は 0 として実効アドレスが求められます。

書式：　(*bd*,A*n*,R*n*.*size***scale*)
bd　：16 または 32 ビットのベースディスプレースメント
size　：インデックスレジスタのサイズ指定 (w または l)
scale　：スケール値指定 (1，2，4，8 のいずれか)

4) 16 ビットの場合は 32 ビットに符号拡張が行われます。

5) データレジスタまたはアドレスレジスタです。ワードまたはロングワードのサイズ指定が必要です。ワードサイズが指定されると，レジスタの内容の下位 16 ビットが 32 ビットに符号拡張されます。

A-1-8 メモリ間接形式

これは 68020 以降になって追加されたアドレス形式です。実効アドレスを求めるために，メモリ上の値が使用されます。

◆ ポストインデックスつきメモリ間接形式

まず，アドレスレジスタ A*n* の内容と，16 ビットまたは 32 ビットのベースディスプレースメント *bd* を加えることで中間間接メモリアドレスを求めます。そして，このアドレスにあるロングワードの値とインデックスとして指定したレジスタ R*n* の内容とスケール値 *scale* との積，および 16 ビットまたは 32 ビットの**アウタディスプレースメント** *od* を加えた値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。
このアドレス形式では，アドレスレジスタ，インデックスレジスタ，ベースディスプレースメント，アウタディスプレースメントのすべてをそれぞれ任意に省略することができます。省略した値は 0 として実効アドレスが求められます。

書式：　([*bd*,A*n*],R*n*.*size***scale*,*od*)
bd　：16 または 32 ビットのベースディスプレースメント
size　：インデックスレジスタのサイズ指定 (w または l)
scale　：スケール値指定 (1, 2, 4, 8 のいずれか)
od　：16 または 32 ビットのアウタディスプレースメント

◆ **プリインデックスつきメモリ間接形式**

まずアドレスレジスタ A*n* の内容と，16 ビットまたは 32 ビットのベースディスプレースメント *bd*，そしてインデックスとして指定したレジスタ R*n* の内容とスケール値 *scale* との積をすべて加えることで中間間接メモリアドレスを求めます。そして，このアドレスにあるロングワードの値と 16 ビットまたは 32 ビットのアウタディスプレースメント *od* を加えた値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。

このアドレス形式では，アドレスレジスタ，インデックスレジスタ，ベースディスプレースメント，アウタディスプレースメントのすべてをそれぞれ任意に省略することができます。省略した値は 0 として実効アドレスが求められます。

書式：　([*bd*,A*n*,R*n*.*size***scale*],*od*)

bd　：16 または 32 ビットのベースディスプレースメント

size　：インデックスレジスタのサイズ指定 (w または l)

scale　：スケール値指定 (1，2，4，8 のいずれか)

od　：16 または 32 ビットのアウタディスプレースメント

A-1-9　ディスプレースメントつきプログラムカウンタ間接形式

プログラムカウンタ PC の内容と，16 ビットのディスプレースメント値 *d* (−32768 ～ 32767) を 32 ビットに符号拡張した値を加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます[6]。

書式：　(*d*,PC)

d：16 ビットのディスプレースメント (−32768 ～ 32767)

6) 前著では「プログラムカウンタ相対形式 (***Program Counter relative***)」となっていましたが，68020 以降になって「プログラムカウンタ間接形式 (***Program Counter indirect***)」と表記が変わりました。単に表記が変わっただけで，その内容は同じです。

A-1-10　インデックスつきプログラムカウンタ間接形式

◆ **8 ビットディスプレースメント**

このアドレス形式は，68000，68010 のインデックスつきプログラムカウンタ間接形式の拡張になっています。68020 以降になって，インデックスレジスタに対するスケール値の指定が追加されました。

プログラムカウンタ PC の内容と，8 ビットのディスプレースメント値 *d* (−128 ～ 127) を 32 ビットに符号拡張した値，そしてインデックスとして指定したレジスタ R*n*[7] の内容とスケール値 *scale* との積をすべて加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。

インデックスレジスタ R*n* には，ワードまたはロングワードのサイズ指定

7) データレジスタまたはアドレスレジスタです。

が必要です。ワードサイズを指定すると，レジスタの内容の下位 16 ビットを 32 ビットに符号拡張した値とスケール値との積がアドレス指定に使用されます。

書式： (*d*,PC,R*n.size***scale*)

d ：8 ビットのディスプレースメント (−128 ～ 127)

size ：インデックスレジスタのサイズ指定 (w または l)

scale ：スケール値指定 (1, 2, 4, 8 のいずれか)

◆ベースディスプレースメント

このアドレス形式は，従来のインデックスつきプログラムカウンタ間接形式のディスプレースメントを 16 または 32 ビットに広げる拡張が行われたものです。

プログラムカウンタ PC の内容と，16 ビットまたは 32 ビットのベースディスプレースメント *bd*[8]，そしてインデックスとして指定したレジスタ R*n*[9]の内容とスケール値 *scale* との積をすべて加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。

このアドレス形式では，プログラムカウンタ，ベースディスプレースメント，インデックスレジスタのそれぞれを任意に省略することができ，省略した値は 0 として実効アドレスが求められます。ただし，プログラムカウンタを省略する場合には，実効アドレスがプログラム空間参照であることを表わすために，疑似レジスタ名 “ZPC” を指定する必要があります。

書式： (*bd*,PC,R*n.size***scale*)

bd ：16 または 32 ビットのベースディスプレースメント

size ：インデックスレジスタのサイズ指定 (w または l)

scale ：スケール値指定 (1, 2, 4, 8 のいずれか)

8) 16 ビットの場合は 32 ビットに符号拡張が行われます。

9) データレジスタまたはアドレスレジスタです。ワードまたはロングワードのサイズ指定が必要です。ワードサイズが指定されると，レジスタの内容の下位 16 ビットが 32 ビットに符号拡張されます。

A-1-11 プログラムカウンタメモリ間接形式

これは 68020 以降になって追加されたアドレス形式です。実効アドレスを求めるために，メモリ上の値が使用されます。

◆ ポストインデックスつきプログラムカウンタメモリ間接形式

まずプログラムカウンタ PC の内容と，16 ビットまたは 32 ビットのベースディスプレースメント *bd* を加えることで中間間接メモリアドレスを求めます。そして，このアドレスにあるロングワードの値とインデックスとして指定したレジスタ R*n* の内容とスケール値 *scale* との積，および 16 ビットまたは 32 ビットのアウタディスプレースメント *od* を加えた値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容

に対して行われます。
このアドレス形式では，プログラムカウンタ，インデックスレジスタ，ベースディスプレースメント，アウタディスプレースメントのすべてをそれぞれ任意に省略することができます。省略した値は 0 として実効アドレスが求められます。ただし，プログラムカウンタを省略する場合には，実効アドレスがプログラム空間参照であることを表わすために，疑似レジスタ名 “ZPC” を指定する必要があります。

書式： ([*bd*,PC],R*n*.*size***scale*,*od*)

bd　：16 または 32 ビットのベースディスプレースメント
size　：インデックスレジスタのサイズ指定 (w または l)
scale　：スケール値指定 (1, 2, 4, 8 のいずれか)
od　：16 または 32 ビットのアウタディスプレースメント

◆ **プリインデックスつきプログラムカウンタメモリ間接形式**

まずプログラムカウンタ PC の内容と，16 ビットまたは 32 ビットのベースディスプレースメント *bd*, そしてインデックスとして指定したレジスタ R*n* の内容とスケール値 *scale* との積をすべて加えることで中間間接メモリアドレスを求めます。そして，このアドレスにあるロングワードの値と 16 ビットまたは 32 ビットのアウタディスプレースメント *od* を加えた値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。
このアドレス形式では，プログラムカウンタ，インデックスレジスタ，ベースディスプレースメント，アウタディスプレースメントのすべてをそれぞれ任意に省略することができます。省略した値は 0 として実効アドレスが求められます，プログラムカウンタを省略する場合には，実効アドレスがプログラム空間参照であることを表わすために，疑似レジスタ名 “ZPC” を指定する必要があります。

書式： ([*bd*,PC,R*n*.*size***scale*],*od*)

bd　：16 または 32 ビットのベースディスプレースメント
size　：インデックスレジスタのサイズ指定 (w または l)
scale　：スケール値指定 (1, 2, 4, 8 のいずれか)
od　：16 または 32 ビットのアウタディスプレースメント

A-1-12 絶対ショートアドレス形式

実効アドレスには，オペランドに指定した 16 ビットの絶対アドレスを 32 ビットに符号拡張した値が使用されます。したがって，指定可能なアドレスの範囲は $00000000 から $00007FFF および $FFFF8000 から $FFFFFFFF に限られます。

書式： <式> .w

A-1-13　絶対ロングアドレス形式

実効アドレスには，オペランドに指定した32ビットの絶対アドレスが直接使用されます。

指定したアドレスにサイズ指定がない場合には，式の値によって絶対ショートアドレス形式と絶対ロングアドレス形式を自動的に選択します。

書式：　<式>.l

A-1-14　イミディエイト形式

命令の操作の対象は，オペランドに指定した値そのものとなります。指定する値の先頭には，イミディエイト形式であることを示すために“#”を置きます。

書式：　# <式>

A-2 アドレス形式指定の注意点

68020 以降の CPU で拡張されたアドレス形式は，レジスタ指定が省略できるようになったことなどからアドレス指定の柔軟性が向上しましたが，そのためにまぎらわしい表記になることが多くなっています。最後に，こうしたアドレス形式指定のうえでの注意点をあげておきます。

A-2-1 従来のアドレス形式との重複

拡張されたアドレス形式では，レジスタやディスプレースメントの指定を自由に省略できるようになったため，省略によって従来からあるアドレス形式と同じものになってしまうことがあります。このような場合，アセンブラはよりサイズの小さい，実行速度の速い命令を得るために，従来からのアドレス形式で表記できるものは必ずそちらを選択します。

List A-1 の 1 行目は単なるアドレスレジスタ間接形式ですが，「拡張されたインデックスつきアドレスレジスタ間接形式から，ベースディスプレースメントとインデックスレジスタを省略したもの」と見ることもできます。ここではアセンブラはアドレスレジスタ間接形式として扱います。

2 行目も同様に，ディスプレースメントつきアドレスレジスタ間接形式とも，インデックスつきアドレスレジスタ間接形式からインデックスレジスタを省略したものとも見ることができますが，従来からあるアドレス形式である，ディスプレースメントつきアドレスレジスタ間接形式として扱います。

ところが，3 行目ではディスプレースメントの値 $12345 が 16 ビットの範囲を超えているため，従来のアドレス形式では扱うことができません[1]。そこで，これはインデックスつきアドレスレジスタ間接形式からインデックスレジスタが省略されたものとして扱うことになります。

1) ディスプレースメントつきアドレスレジスタ間接形式で扱えるディスプレースメント値は，16 ビットの範囲内 (−32768～32767) に制限されます。

List A 1 従来のアドレス形式との重複

```
1:      (a0)
2:      ($1234,a1)
3:      ($12345,a1)
```

このような省略によって，ディスプレースメントつきアドレスレジスタ間接形式のディスプレースメントが32ビットに拡張されたかのように使用することができるわけです。

A-2-2 インデックスサイズの省略

インデックスつきアドレスレジスタ間接などのアドレス形式で，インデックスレジスタとして指定するレジスタは，サイズ指定を省略するとワードサイズとして扱われます[2)]。ところが，ベースレジスタとして使用するアドレスレジスタやプログラムカウンタはサイズを指定せず[3)]，ロングワードの値全体が使用されるため，一見して分かりにくいアドレス指定になってしまうことがあります。

2) これは68000や68010でも同じです。

3) 指定するとエラーになります。

List A-2の1行目にはアドレスレジスタが2つあります。インデックスつきアドレスレジスタ間接形式では，ベースレジスタとなるアドレスレジスタは，インデックスレジスタとなるレジスタより先に指定することになっているため，この例ではA1レジスタがベースレジスタ，A5レジスタがインデックスレジスタということになります。つまり，A1レジスタはロングワード値全体を使用し，A5レジスタは下位ワードを符号拡張して使用することになるわけです。

また，2行目はさらに分かりにくい例です。インデックスつきアドレスレジスタ間接形式からベースレジスタとベースディスプレースメントを省略するとインデックスレジスタだけが残りますが，これにD0レジスタを指定すると，この例のように，一見「データレジスタ間接形式」であるかのような表記ができます。ところが，このレジスタはサイズ指定が省略されたインデックスレジスタなので，アドレス指定には下位ワードを符号拡張した値が使われてしまいます[4)]。

4) おそらく，これは表記の意図とは異なった結果となることでしょう。

List A 2 インデックスサイズの省略

```
1:      (a1,a5)
2:      (d0)
```

このように，インデックスレジスタのサイズを省略すると，アドレス形式の表記が分かりにくくなってしまうことが多くなります。そのため，**X680x0 HAS**では，インデックスレジスタのサイズが省略された場合に，“**index size not specified**”というワーニングを出力するようになっています[5)]。

5) Chapter 2「変更された診断メッセージ」(P.48)を参照してください。

A-2-3 X680x0 HAS の独自表記

アドレス形式を指定するうえで，**X680x0 HAS** が特別に許可している表記がいくつかあります。これらは独自の表記であるため，**X680x0 HAS** 以外のアセンブラでは使用できない場合があることを注意してください。

◆ アドレス指定要素の表記順

List A-3 は，ポストインデックスつきメモリ間接形式の指定例です。このアドレス形式では，

- **ベースディスプレースメント**
- **ベースレジスタとなるアドレスレジスタ**
- **インデックスレジスタ**
- **アウタディスプレースメント**

の 4 つをこの順序で，1 行目のように表記することになっていますが，**X680x0 HAS** では 2 行目と 3 行目のように，内容が同じであれば順序は制限されません。これらはみな，同じ実効アドレスを表します。

List A 3 アドレス指定要素の表記順

```
1:      ([$abcd,a0],d2.w*8,$3456)
2:      ($3456,[$abcd,a0],d2.w*8)
3:      (d2.w*8,$3456,[a0,$abcd])
```

◆ レジスタ名の省略

List A-4 は，インデックスつきアドレスレジスタ間接形式からベースレジスタとなるアドレスレジスタを省略した例です。

1 行目が本来の表記なのですが，**X680x0 HAS** では 2 行目のように，レジスタを省略することを明記するために，レジスタ名の代わりに先頭に “Z” をつけた疑似レジスタ名 (ZA0 ～ ZA7，ZD0 ～ ZD7) を指定することもできます。

省略したレジスタの値は使用されないので，実際にはどのレジスタを指定しても実行結果は変わりません。この例では，2 行目と 3 行目はまったく同じ動作になります。

List A 4 レジスタ名の省略

```
1:      ($1234,d0.l*2)
2:      ($1234,za0,d0.l*2)
3:      ($1234,za2,d0.l*2)
```

Chapter 9

Appendix B

本セクションでは，各ツールのオプションスイッチと診断メッセージ，アセンブラ擬似命令一覧，GDB コマンド一覧，ライブラリ関数機能別一覧を掲載します。
各コマンドの右端に I と記載されているのは「*X68000 Develop.*」「同 *Manual Books*」または「*X680x0 libc*」「同 *Manual Books*」のページ番号を，II と記載されているのは本書でのページ番号を指しています。
また，前著と新著とでページ番号が異なる場合は，新著のページ番号をカッコ内に記載しました。

B-1 各ツールオプション一覧

B-1-1 GCCのオプションスイッチ

○ コンパイラドライバのオプションスイッチ

○ X680x0 GCC 独自のオプションスイッチ

B-1-2 HASのオプションスイッチ

B-1-3 HLKのオプションスイッチ

B-1-4 GDBのオプションスイッチ

B-1-5 LIBCのオプションスイッチ

B-2 診断メッセージ一覧

B-2-1 GCCのメッセージ

○ エラーメッセージ

○ ワーニングメッセージ

B-2-2 HASのメッセージ

○ エラーメッセージ

○ ワーニングメッセージ

B-2-3 HLKのメッセージ

B-2-4 LIBCのメッセージ

B-3 アセンブラ擬似命令一覧

B-3-1 アセンブラ制御

B-3-2 セクション指定

B-3-3 外部名の宣言

B-3-4 シンボルの定義

B-3-5 マクロ制御

B-3-6 データの定義

B-3-7 条件つきアセンブル

B-3-8 リストファイル制御

B-3-9 シンボリックデバッグ情報の指定

B-4 GDB コマンド一覧

B-4-1 実行を制御するコマンド

B-4-2 スタックフレームを調査するコマンド

B-4-3 データに関するコマンド

B-4-4 ブレークポイントに関するコマンド

B-4-5 ファイルに関するコマンド

B-4-6 プログラムの状態を調査するコマンド

B-4-7 infoコマンドのサブコマンド

B-4-8 maintenance コマンドのサブコマンド

B-4-9 GDB を設定するコマンド

B-5 ライブラリ関数機能別一覧

B-5-1 C標準関数

○ 数値演算

○ プロセスの環境設定・取得

○ コンソール直接入出力

○ 文字の判定と変換

○ 低水準ファイル入出力とファイル名操作

○ プロセス操作

○ シグナル操作

○ ソート

○ 標準ファイル入出力

○ 文字列とメモリ領域の操作

○ 時間の取得と設定

○ ユーザデバイス管理

○ その他

B-5-2 DOS コール

B-5-3 IOCSコール

B-5-4 マルチバイト文字

B-5-5 SCSI コール

B-5-6 幅広文字

索　引

Q

R

S

T

V

W

Z

あ行

か行

さ行

た行

な行

は行

ま行

や行

ら行

わ行

X68k Programming Series #0

X680x0 Develop. & libc II

1994年9月5日　初版発行

著　者　吉野智興　中村祐一　石丸敏弘　今野幸義
村上敬一郎　大西恵司

発行者　橋本五郎

発行所　ソフトバンク株式会社出版事業部
〒103 東京都中央区日本橋浜町 3–42–3
TEL　営業部 03 (5642) 8101
編集部 03 (5642) 8143

写植・印刷　東京書籍印刷株式会社

ISBN4-89052-535-1

Design = Tetzya Yonetani

SOFT BANK ソフトバンク

ISBN4-89052-535-1

C0055 P2900E

9784890525355

定価2,900円
（本体2,816円）

1910055029002

X68k
Programming Series
(#0)
X680x0
Develop. & libc II